30841

NATIONAL LIBRARY
OTTAWA

CANADA

BIBLIOTHÈQUE NATIONALE
OTTAWA

NAME OF AUTHOR.....Steven.....Sutphen.....

TITLE OF THESIS.Virtual.....memory...Experiments
.on...the....UNIX....system...
.....

UNIVERSITY..of....Alberta.....

DEGREE FOR WHICH THESIS WAS PRESENTED..M.Sc.....

YEAR THIS DEGREE GRANTED...1976.....

Permission is hereby granted to THE NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(Signed).Steven Sutphen.....

PERMANENT ADDRESS:

C/o...R.F....Sutphen
.120.6.1...Hwy..410.
.Naches,.Wash.,..98937

DATED July...23....1976

NL-91 (10-68)

INFORMATION TO USERS

THIS DISSERTATION HAS BEEN
MICROFILMED EXACTLY AS RECEIVED

This copy was produced from a microfiche copy of the original document. The quality of the copy is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

PLEASE NOTE: Some pages may have indistinct print. Filmed as received.

Canadian Theses Division
Cataloguing Branch
National Library of Canada
Ottawa, Canada K1A ON4

AVIS AUX USAGERS

LA THESE A ETE MICROFILMEE
TELLE QUE NOUS L'AVONS RECUE

Cette copie a été faite à partir d'une microfiche du document original. La qualité de la copie dépend grandement de la qualité de la thèse soumise pour le microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

NOTA BENE: La qualité d'impression de certaines pages peut laisser à désirer. Microfilmee telle que nous l'avons reçue.

Division des thèses canadiennes
Direction du catalogage
Bibliothèque nationale du Canada
Ottawa, Canada K1A ON4

THE UNIVERSITY OF ALBERTA

VIRTUAL MEMORY EXPERIMENTS ON THE UNIX SYSTEM

by

© STEVEN F. SUTPHEN

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE

OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA

FALL, 1976

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled "Virtual Memory Experiments on the UNIX System", submitted by Steven Forest Sutphen in partial fulfilment of the requirements for the degree of Master of Science.

T. A. Marsland
.....
Supervisor

Barry J. Mailloux
.....

W. K. Dawson
.....

Dated ..... 18 May ..... 1976.

Abstract

An algorithm is developed to convert UNIX from a swapping system to a paging system. The algorithm is broken down into four steps consisting of: separation of the instruction and data address spaces, providing for discontinuous processes in physical mainstore, paging out only those portions of a process which have changed since last written to backing storage, and finally allowing partially loaded processes to run. Each of these stages has advantages and disadvantages. The routines as well as the data structures which need to be modified are described along with the modifications. An important concept, held throughout these experiments, is that at the conclusion of implementing each stage, the system will be usable and should be tested before implementing the following step. Another unifying constraint is that the modifications are carefully planned, that is, they were not implemented and then written about but vice versa. Since UNIX is a well-structured, clean operating system the above technique aids in keeping the system clean and well-structured. To aid in understanding the changes described, as well as their need, detailed discussion of the hardware capabilities of the PDP 11/45 is given, as well as a discussion of the current system software. An abstract, and cross-reference of each routine in the operating system is provided in the Appendix.

Acknowledgement

I would like to thank my supervisor, Dr. T.A. Marsland, for his support and guidance throughout the preparation of this thesis.

I acknowledge the support provided me by the Department of Computing Science, in the form of a teaching assistantship.

Many thanks to the innumerable graduate students here at the University of Alberta, who have provided healthy discussions over the years; thanks also to Ken Thompson of Bell Laboratories, who provided the basis for this work.

Table of Contents

I Introduction..... 1

II The Software..... 5

2.1 Introduction..... 5

2.2 In The Beginning..... 6

2.3 Growing Up..... 14

2.4 Transmogrification..... 18

2.5 Making Friends..... 20

2.6 Waiting in the Wings..... 23

2.7 Departing This World..... 25

III The Hardware..... 28

3.1 Introduction..... 28

3.2 Processor Modes..... 29

3.3 Relocation..... 35

3.4 Page Descriptor Registers (PDR)..... 39

3.4.1 Access Control..... 39

3.4.2 Statistics Information..... 42

IV The New Software..... 44

4.1 Introduction..... 44

4.2 Stage I Separate Instruction and Data Spaces..... 45

4.3 Stage II Discontinuity of User Processes..... 51

4.4 Stage III Paging Processes..... 56

4.5 Stage IV Paging Segments 62

V Conclusion..... 69

5.1 Results..... 69

5.2 Further Research..... 71

References..... 75

Appendix 1 - System Structures 77

Appendix 2 - Kernel Routine Abstracts 82

List of Figures

1. Flow of the Life of a Process 7

2. Physical Memory for Two Processes 12

3. The User-Structure 13

4. Construction of a Physical Address 38

5. Page Descriptor Register (PDR) 41

Chapter I

Introduction

UNIX is a contemporary operating system which runs on a series of modern mini/midicomputers. It is a "general-purpose, multi-user, interactive operating system for the Digital Equipment Corporation PDP 11/40, 11/45 and 11/70 computers" (Ritchie 1974). The prime intent of the thesis is to explore the memory-management aspects of UNIX. Chapter II examines UNIX as it currently exists, Chapter III explores the hardware that it runs on, and Chapter IV sets forth several experiments which could be performed on UNIX to transform it from a swapping system to a paging system.

In order to provide a sound basis for discussion of certain operating-system concepts, it is necessary first to introduce and define some of the contemporary terminology. In UNIX an 'image' is defined as a computer execution environment. It includes the memory contents, the current general registers, the status of open files, the current directory and the like. An image is the current state of a 'virtual-computer'. Normally there are multiple virtual-computers in main store and perhaps a few on backing store (i.e. swapped out). The real machine, as it is a uniprocessor, is capable of executing only one of these images at a time. The execution of an image is called a 'process'. While the processor is executing on behalf of a process the image must reside completely in main store. During the execution of other processes the image will remain in main store unless an ac-

tive, higher-priority process forces it to be swapped (in its entirety) to backing store (usually a fixed-head disk). The above is a conceptual definition of a 'swapping system', that is, a swapping system requires an image to be completely loaded in main store or completely unloaded onto backing store. A 'paging system', on the other hand, requires only the active portion (or working set Denning (1968)) of an image be loaded in main store for the execution of it to proceed. A discussion of the above terms, and an overview of memory management techniques, are located in Madnick (1974).

The user's portion of an image is divided into three logical pieces. The program text (or instruction) segment begins at location zero of the virtual address space. During execution, the text segment may optionally be write-protected, in which case a single copy of it is shared among all processes executing the same program. At the first 8K byte ('K' used in this way will mean 1024 items throughout the thesis) boundary above the text segment in the virtual address space begins the data segment. It is writable as well as readable, and can be expanded by a system call. At the highest virtual address possible (64K bytes) starts the stack segment. It expands automatically downward as the hardware's stack pointer fluctuates.*

To do input/output, manage files, find the time of day, or to perform various other activities, a process must call

---

* The above two paragraphs were paraphrased from Ritchie (1974) to maintain consistency in the definitions.

on the supervisory portion of UNIX. These calls are termed 'system calls' (implemented via the SYS instruction*) and they call on the 'kernel' of UNIX. The kernel consists of the code and data which are required to interface between the hardware and the user software. It performs several functions, including scheduling the CPU, swapping user processes, loading new processes, maintaining a clock queue, implementing a file system on all disks, and acting as an interface to the various input/output devices.

All entries to the kernel (except one during the bootstrapping) are through a hardware mechanism called a 'trap'. Each input/output device, as well as each type of CPU exceptional condition, has its own unique trap vector. The operation of a trap is as follows:

1). Push the old program counter (PC-register 7; designated R7), and the old processor status word (PS) onto the new (kernel-mode) stack (pointed to by R6, the stack-pointer(SP)).

2). Pick up new PC, PS from trap vector in the kernel-mode data space.

In the above discussion a new term was introduced, 'kernel-mode'. PDP 11/45's and 11/70's are capable of executing in three modes: kernel, supervisor and user (PDP 11/40's have only kernel and user modes). Although the kernel (nucleus) of UNIX always executes in kernel-mode the two terms (kernel and kernel-mode) have different connotations and the latter

* SYS is another name for the TRAP instruction.

is always specified with a hyphen in the thesis.

The current (and previous) mode of the CPU is kept in the PS. Since a new PS is loaded during a trap, the mode of the CPU can be switched through the trap mechanism. In UNIX, all exceptional conditions trap into kernel-mode, where they are handled by various kernel routines. As the old PS and old PC were saved by the interrupt, the machine may be restored to its previous state.

One further important aspect of UNIX, which is essential to understanding the thesis, is that there are no 'control blocks' depended on by system calls which are partially maintained by or contained in the user process. Generally speaking, the contents of a user process' address space are the property of that process and few restrictions are placed on the data structures within that address space. Contrasting systems, those which do have control blocks in the user space, include DEC-DOS11, Digital (1971) and OS/360, IBM (1973).

Since the UNIX operating system is inadequately described elsewhere the thesis must provide one. The following chapter is a detailed description of the portions of UNIX which are relevant to the thesis. Chapter III examines the memory management hardware of a PDP 11/45 and how it is used by UNIX. Chapter IV presents a number of experiments which could be performed on UNIX, given the available hardware. Finally, results and several suggestions for further research are stated in Chapter V.

Chapter II

The Software

2.1 Introduction

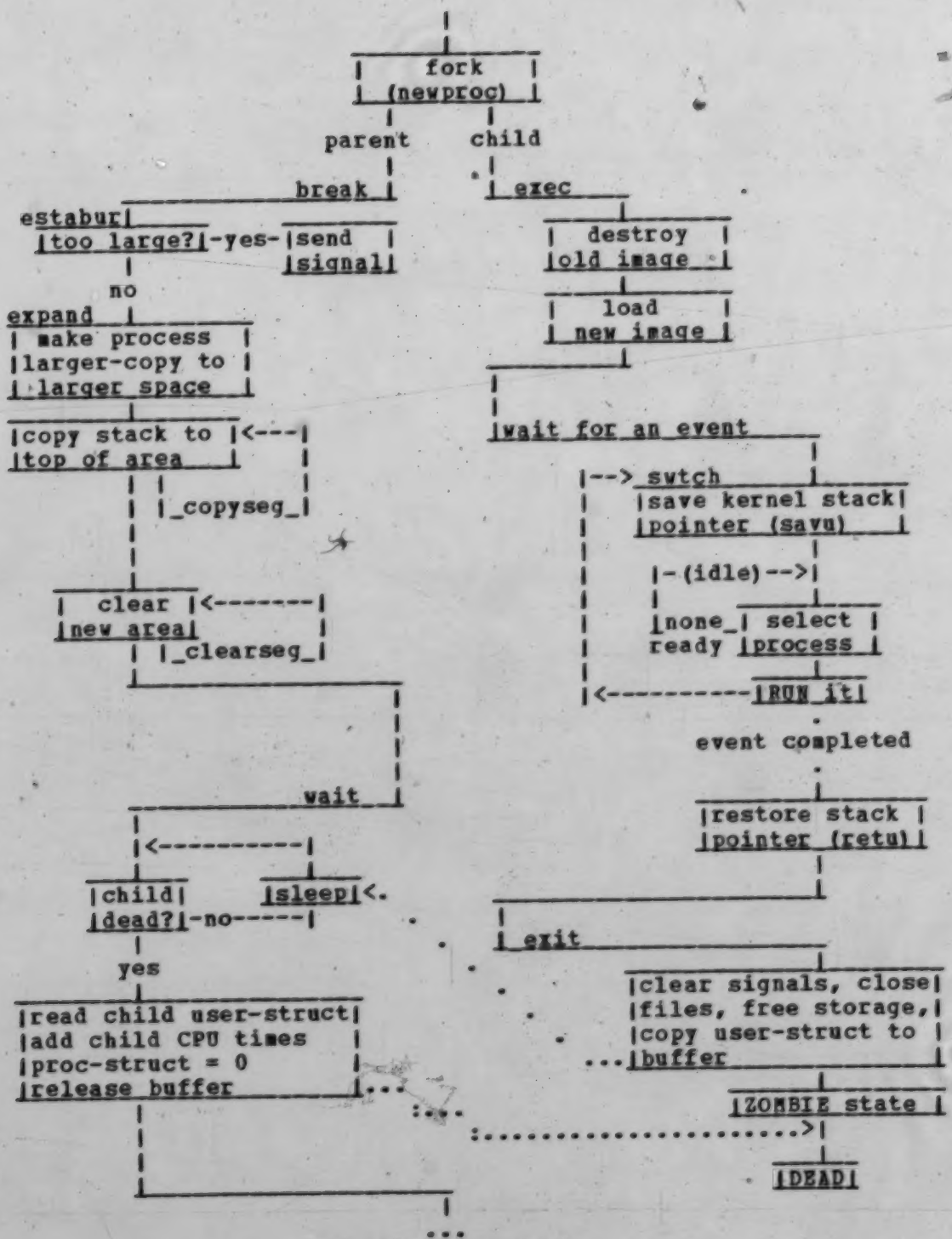
A necessary prerequisite to the appreciation of the system designer's concept of UNIX (as distinct from a user's viewpoint) is a thorough familiarity with the introduction to UNIX, Ritchie (1974), since it provides an overall view of the system and a necessary background to the features discussed here after. UNIX has grown from a humble birth on a PDP7 to an elegant useful system on PDP11/40's, 45's and 70's. The main goals of the system designers were to provide "a comfortable relationship with the machine and with exploring ideas and inventions exploitation in operating systems", Ritchie (1974). Because of these goals, and the inclusion of selected notions from other systems, UNIX is an interesting subject for study by system designers. Various elements of the system which relate to Chapter IV, where some experiments in swapping/paging are developed, are studied here. Only the pertinent modules will be examined. The elements not discussed are the file subsystem, the operation of individual input/output drivers, and several simple utility functions (such as fetching the current time and date).

The chapter is organized around a 'day in the life' of a process, starting from the moment of its conception, through its birth, transfiguration, and death. The transcending functions of memory management, scheduling, and

swapping to and from backing store are covered. Items discussed include the structures involved in each function and other subroutines called upon to implement the primitive. Figure 1 is provided to aid in the description of the life of a process. The centerline of the figure represents the process executing in normal user-mode, while the various side trips represent the tasks performed by the kernel. The dotted arrows indicate actions which cause state transitions. For example, the child entering the ZOMBIE state causes the parent to be awakened. The reader is referred to Appendix 2 for a complete brief abstract and cross reference of the system modules.

2.2 In The Beginning

When the system is initialized (during the bootstrapping operation), the first process is created in an ad hoc way. There are two phases to the first process. At first it creates a second process (referred to as the *init* process), and thereafter it becomes the swapping process. The details of the swapping process are discussed in Section 2.6, and the *init* process is discussed here. The *init* process consists of the execution of the image stored in the file /etc/init. The most relevant function which *init* performs is the creation and maintenance of a process for each terminal on the system. These processes are called the logon listeners, because they listen to the terminals associated with them and wait until someone types in his logon-name. As



Flow Diagram of the
Life of a Process

Figure 1

the creation and execution is typical of all processes (that is, these processes use no special privileges) it would be appropriate to examine the details from an internal, or system view. Actually there are two system primitives issued to arrive at the point of being a logon listener. The first primitive executed is called fork()*, and is described in this section. The other primitive is named exec() and is described in Section 2.4.

When the fork() primitive is executed by a process, two returns are made by the system, one for the original process (the parent) and one for the new process (the child). This operation is generally called sub-tasking or forking. The two processes have independent copies of mainstore and share open files. The only difference between the two returns is the value of the function: for the child the function value is zero, while for the parent it is the process ID of the child. The notion of the fork() primitive was borrowed from the Berkeley Time-Sharing System, Deutsch (1965). The remainder of the section will describe the internal flow needed to obtain two processes from one, or what a process really is to the system.

The first level system primitive handler, trap(), uses the 'primitive number' (operand of the system call) as an index into a table of entry-point addresses (sysent[ ]) of the various primitives. In the case of fork, the entering

* The fork primitive and other primitives mentioned in the thesis are described in "UNIX Programmer's Manual", Thompson (1975).

routine is named, quite appropriately fork(). After checking to see if room in the process table exists for a new process, fork() calls the logical function newproc(). Newproc() is the function which actually turns one process into two, by returning twice, the first time directly with a return value of zero. The second return, for the newly created process, is much more indirect and, since it manipulates important system data structures, it will be discussed in more detail.

The prime function of newproc() is to generate a copy of the data structures which describe a process. The most important structure, and the one from which all other structures may be located is the 'proc-structure'. The process table contains all active and inactive proc-structures. The proc-structure contains the information required to return a process to mainstore after it has been swapped out. The items which are copied from the parent's proc-structure into the child's are: the user id; the controlling terminal descriptor pointer; and a pointer to the descriptor of the sharable text portion of the process. If the sharable text portion exists, the counts of the number of processes currently able to access it on the disk and in main memory are incremented. It is important to note that the mainstore count can safely be incremented because the text is guaranteed to be loaded. The operation of the kernel depends on the text being loaded, as well as several other principles.

The fundamental principles which newproc(), as well as other routines depend on are examined in detail here. First and most important, the kernel routines rely on the fact that their execution is continuous. The term 'continuous execution' does not imply that interrupts such as the clock and other I/O devices are not fielded, but that the execution of the interrupt handlers leaves the kernel process unaffected. That is, it is not possible for a process switch to occur when in kernel-mode (unless the kernel process explicitly asks for a switch by calling swtch()). It is also not possible for a kernel-mode process to be swapped out. Finally it is not possible for the kernel-mode process to be modified by the I/O event because all I/O is synchronous at the process level.

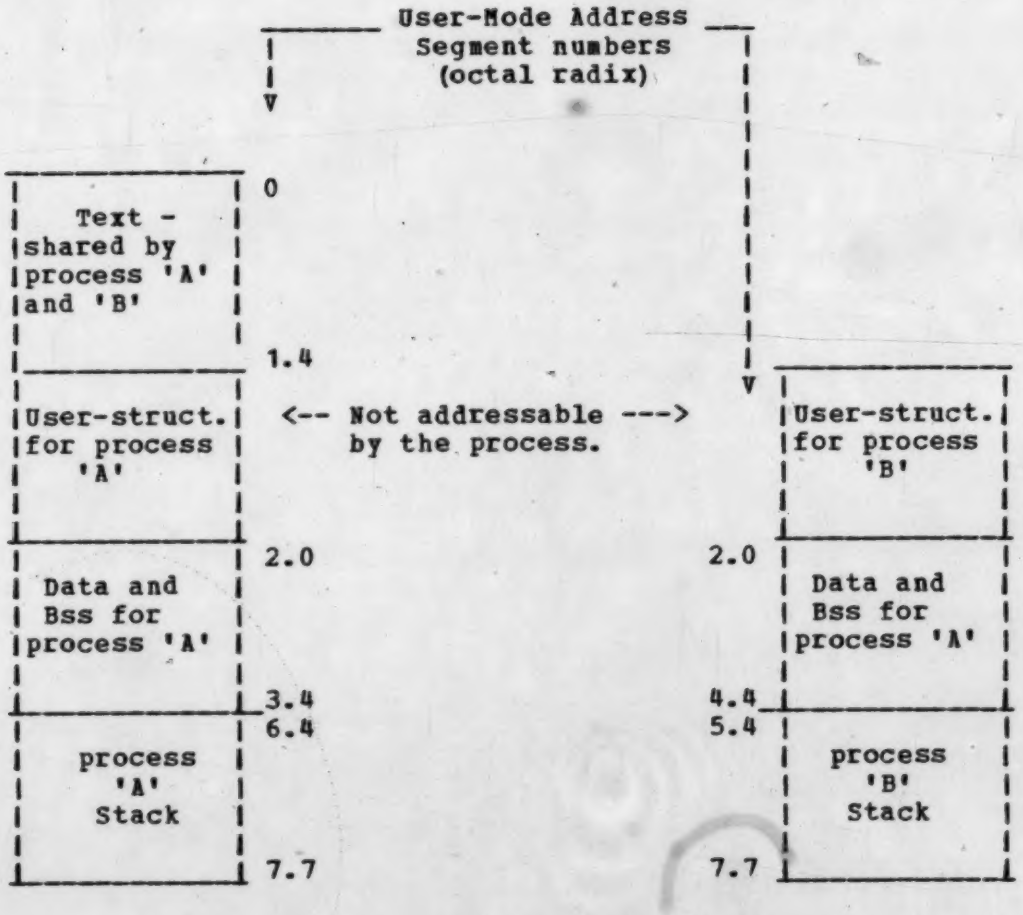
The second fundamental premise upon which newproc() depends is that a process is completely loaded while it is running. Even in kernel-mode the process is completely loaded, and so newproc() may increment the mainstore use count of the text portion of a process.

The final concept which newproc() depends on is the continuity of the physical memory image of a process. The copying mechanism (copyseg()) in newproc() depends on the fact that the 'user-structure' (which is pointed to by p_addr in the proc-structure), data, and stack are physically adjacent in mainstore although the addressing of them in user-mode is quite often discontinuous. For example, in Figure 2 if process 'A' addresses segment 3.4 (address 064000 octal) it

would receive a 'bus error' signal from the system, yet that address physically exists and is accessed by the user-process as segment 6.4 (address 0144000).

The term 'user-structure' introduced in the last paragraph requires some more explanation. This structure contains descriptive information about the process which owns the structure. Since there is one and only one user-structure for each process the structure may be thought of as an extension of the proc-structure. The major reason for placing data items in the user-structure, instead of just having a larger proc-structure, is that the proc-structure is resident, always using up valuable mainstore, while the user-structure is attached to the text and data of the process and is swapped in and out with it. Figure 3 contains the portions of the user-structure which are relevant to the thesis. The complete commented structure is given in Appendix 1, along with a commented proc-structure.

A brief introduction to the PDP 11/45's memory-management unit (MMU) is given here to explain some of the notions of the previous paragraph. The hardware unit which performs memory mapping is referred to as a MMU. It is capable of mapping memory differently depending on the mode of the machine. There are three modes possible on a PDP11/45; kernel-mode - the least restrictive mode; supervisor-mode - restricted instruction set, not used much in UNIX; and the user-mode - the most restricted mode. The normal execution of a process commonly passes between user-mode (while it is



Physical Memory for Two Processes

Sharing a Common Text Segment

Figure 2

```
struct user{
        int     u_rsav[2];      /* save r5,r6 when exchanging
                                   stacks */
        int     u_fsav[25];     /* save fp registers */
        char    u_segflg;       /* flag for IO; user or kernel
                                   address space */
        char    u_error;        /* return error code */
        char    u_uid;          /* effective user id */
        char    u_gid;          /* effective group id */
        char    u_ruid;         /* real user id */
        char    u_rgid;         /* real group id */
        int     u_procp;        /* pointer to proc structure */
        char    *u_base;        /* base address for IO */
        char    *u_count;       /* bytes remaining for IO */
        char    *u_offset[2];   /* offset in file for IO */
        int     u_uisa[8];      /* prototype of segmentation
                                   addresses (PAR's) */
        int     u_uisd[8];      /* prototype of segmentation
                                   descriptors (PDR's) */
        int     u_ofile[NOFILE];/* pointers to file structures
                                   of open files */
        int     u_arg[5];       /* arguments to system call */
        int     u_tsize;        /* text size (*64) */
        int     u_dsize;        /* data size (*64) */
        int     u_ssize;        /* stack size (*64) */
        int     u_qsav[2];      /* label variable for quits and
                                   interrupts */
        int     u_ssav[2];      /* label variable for swapping */
        int     u_signal[NSIG]; /* disposition of signals */
        int     u_utime;        /* this process' user time */
        int     u_stime;        /* this process' system time */
        int     u_cutime[2];    /* sum of children's utimes */
        int     u_cstime[2];    /* sum of children's stimes */
        int     *u_ar0;         /* address of users saved R0 */
/* kernel stack per user extends from u + USIZE*64
 *      backward not to reach here */
} u;
```

User - Structure (relevant subset)
Figure 3

executing its text and data areas) and kernel-mode (to execute system primitives). For each of the three modes there exists a set of mapping registers, referred to as the memory management registers. These registers provide relocation and protection for instruction and data fetches performed by the CPU. For a more complete description of the MMU see Chapter III.

After doing various odd initializations, such as finding a unique process id, incrementing open-file counts, setting the state of the process to RUN/LOADED, and saving the parent process id, two copies of the calling process are made. The user-structure, data section, and stack are copied into a second space, either in mainstore (if there is room) or swapped out to the disk and marked as swapped. Malloc() allocates the memory required (data size + stack size + user-structure size (USIZE)) on a first fit basis, from the list of available space, 'coremap'. If there is not enough main memory, space is allocated on the swapping disk by passing the disk map 'swapmap' to malloc(). Newproc() then returns to the caller, fork(), with a return value of zero to indicate the parent. When the newly created process is scheduled for execution it also returns to fork(), but since it completes the return through swtch() (the process scheduler) it will return a value of one, identifying itself as the child.

2.3 Growing Up

Processes are capable of growing larger via two different methods: firstly through automatic stack expansion, secondly by calling the system primitive break(). The break() primitive differs enough from the automatic stack expansion that it will be discussed separately. As stack expansion is the simpler of the two functions it will be described first.

When the address of a data word is outside the current bounds of the MMU, a memory-management trap is generated by the hardware. After the user's registers are saved, execution is transferred to the 'trap' handler, which determines whether a stack overflow caused the problem. The process' stack pointer is compared with its current minimum value (u_ssize), which is kept in the user-structure. If the stack pointer is less than u_ssize then a stack overflow must have occurred. If no stack overflow occurred an error signal is sent to the running process, to indicate an addressing violation. If the interrupt was caused by a stack overflow the instruction being executed is backed up and additional stack space is allocated before the process is allowed to proceed.

For the sake of efficiency the allocation proceeds in a rather strange order. First the process' user memory management registers are set up by estabur() (establish user registers). Estabur() checks to see whether the additional size can be accommodated. The rules of accommodation are: the text, data and stack portions must all start on segment (4K word) boundaries, must not overlap, and must fit in

eight segments total. If the above rules are not violated the memory-management register skeletons are built in the user-structure under the names 'u_usia' and 'u_usid'. If the process will no longer fit, estabur() will return an error value of minus one, and will set 'u_error' to the 'ENOMEM' error value. Providing estabur() was able to complete its task the routine expand() is called to expand the size of the process. The mechanics of expand() parallel those of newproc() to some extent. If there is room in main memory for a second, larger copy of the process (as determined by malloc()) it is copied to that area. If not, the process is swapped out to disk with the new larger size. When it is swapped back in, a return from expand() will be simulated by swtch(), and the mainstore image will be the correct size. The new space is added physically above user's stack. Since the stack expands downward the new space should be below the current stack. Therefore the stack must be copied to the top of the allocated memory. Successive calls to copyseg(), which will copy thirty-two words of physical memory each call, are used to move the stack to its proper place. Finally the last important act which trap() performs is to set the new stack area to zero, so the user may not read the garbage (private-protected? data) left in memory. Successive calls to clearseg(), which clears thirty-two words of physical memory each call, are used to overwrite the sensitive data. Because of the large overhead in expanding the stack, it is increased by twenty 32-word chunks

(i.e. SINCR = 640 words) each time the stack overflows.

The second method of changing the size of a process' memory image is the break() primitive, which is capable of both expansion and contraction. The break() primitive is sbreak() in the kernel to avoid disastrous confusion with the break operator in the 'C' language. After fetching the parameter (from u.u_arg[0]), which is the new data size required, estabur() is called to determine if the process will fit in the limits of the MMU. If not, an immediate return is made. The data-size variable 'u.u_dsize' is then replaced by the parameter and the algorithm splits into two streams depending on whether the process is expanding or contracting. If it is growing larger the steps followed are the same as in the stack expansion algorithm explained above: expand() is called to get more space added above the stack; copyseg() is called to copy the stack up to the top of allocated space; and clearseg() is called to clear the new space. If the process is getting smaller, the elements of the algorithm are almost reversed. First the stack is copied down onto the data area which is to be freed. Then expand() is called to free up the top of memory where part of the stack was. Expand() simply calls mfree() to add the freed space to 'coremap', the mainstore free storage map.

Memory may be added only at the top of the data space, or at the bottom of the stack space. Also, memory may be released only from the top of the data area. These two facts make dynamic allocation/deallocation of data areas clumsier

than is usually desirable.

2.4 Transmogrification

The major system primitive exec(file, arg1, arg2, ..., argn) requests the system to load the file named by 'file', and to commence its execution. Execution always starts at virtual location zero. The arguments are passed on to the newly created memory image on the newly created stack. The calling process' memory image is overlayed by the new file's image, and is completely lost. All open files and 'pipes' remain open and unchanged across exec() calls. Pipes are input/output devices which consist simply of software buffers. See Ritchie (1974) for a complete explanation of pipes. Ignored signals remain ignored across exec() calls, but caught signals are reset to their default values. Because the catching routine will no longer exist after the completion of the exec() primitive, caught signals must be reset. A brief explanation of the exec() routine will give a better understanding of the data structures manipulated during the destruction of the old process image, and the loading of the new one.

The first task is to determine whether the file to be loaded exists and is executable by the calling user. If the test is unsuccessful exec() will return to the caller. Otherwise, the arguments to the new process image are copied into a disk buffer and then the first four words of the file are read in. These four words contain the type flag, text

size, data size, and bss size respectively. Estabur() is called to check whether the specified file will fit in 32K words, the total virtual address space of a user program. If the preceding check is affirmative a final commitment is made to load the new image; otherwise, an error return is made. Before the new image is loaded the old one must be unloaded. A call to xfree() is made to free up the process' possession of a text segment. All of the mainstore of the calling process is then freed by calling expand() with the size of the user-structure, thereby freeing everything but the structure.

The text portion of the new process image is loaded first. Routine xalloc() is called to find the text for the new process image. There are four scenarios followed to find the text. First and simplest, is that no text portion exists for the new process image. In this case xalloc() simply returns. The second and third cases are related in that both require the text to be in use by another process. In the second case not only is another process using it, but it is currently loaded. Thus xalloc() simply increments the memory and use counts in the descriptor of the text, and places a pointer to the descriptor in the proc-structure for the process. If the text is not loaded it must first be loaded and the counts then incremented. The fourth case entails a bit more work. No process is using the text, and therefore a copy of it does not exist on the swapping disk. Xalloc() calls expand() (which may have to swap the process

out and back in) to generate a region for the text portion. After the text is read into the newly generated region from the file being loaded, the text is swapped out to disk. Finally xswap() is called to swap the text back into mainstore in the proper place and with the proper flags set.

Once the text has been established properly in memory the new process image is expanded to its final size, the total of the user-structure, data size, and initial stack size. The data and stack areas are initialized to zero, and the initialized data is read into the data area. Finally the parameters to the new image are copied onto the stack, and various housekeeping items (such as setting the user id, clearing signals, and clearing registers) are completed. The primitive then exits to the new process image starting at location zero, because the program counter (register seven) was cleared.

The net effect of the freeing, allocating, clearing, and copying is to install a new process image in mainstore. In section 2.6 the swapping portion of the above algorithm will be discussed in more detail.

2.5 Making Friends

This section deals with the scheduling of the CPU among the awaiting processes. The process-switching routine swtch() is called from various routines in the system, including the clock handler, the trap() handler, and other system routines which wish to allow an event to occur before

returning to their caller. To schedule the central processor among active processes, the clock handler calls swtch() every second, provided the processor was in the user-mode prior to the clock interrupt. The important fact to note here is that the system will not do a process switch when executing system code unless that code explicitly asks it to do so. The uninterrupted execution of the kernel is extremely basic to the structure of the system, and removes much of the need for locking critical sections of code. If a user program spends a large percentage of its time executing system primitives the probability that the clock will tick while it is in user-mode is quite small. Since it is not desirable in a timesharing environment for a single process to monopolize the processor, the trap handler trap() calls swtch() every fifteenth system primitive executed by one process. The other calls to swtch() are made by a terminating process, or by subroutines which must wait idly for an event. Some typical events include completion of input/output, completion of a memory swap, or waiting for a prearranged period of time to elapse.

Now that the means of entering swtch() have been established, an explanation of its execution is in order. The first item attended to is the saving of the kernel-stack level in the user-structure, via a call to savu(). When the process is rescheduled for execution these registers will be restored by retu() to return the kernel-stack to its previous state. As the old process is to be abandoned, a tem-

porary stack is set up for swtch(), in user-structure of process zero (the scheduler's, see next section). A new process is selected from the process table on a highest priority, round robin basis. The priority of a process is determined by two factors. First if the process is in user-mode its priority is the system constant 100, plus a user alterable variable u.u_nice in the range of 0 to 20. The higher the priority value the lower the priority. If a system primitive has been waiting for an event then the priority of the process for which the primitive is being executed is determined fairly arbitrarily by the waiting routine. These priorities are usually quite high (for example, while waiting for a swap to complete, the priority is set to -100). If there are no processes wishing to be run, the processor will be placed in the wait state via a call to idle(). Once a process has been selected to be run, its user-structure is placed (through manipulation of the memory-management registers) at location 0140000, and the kernel's stack pointer is returned from the user-structure via retu(). The memory-management registers for the new user process are built from the skeleton registers in the user-structure. The skeleton was placed in the user-structure previously by estabur(). Finally swtch() returns, which indirectly restarts the user process where it left off. Swtch() will schedule only loaded processes for processor usage, and therefore needs an independent agent to load and unload processes from the swapping medium. This

agent is described in the next section.

2.6 Waiting in the Wings

The scheduler is the only process which executes in the kernel-mode all of its life. It is also the first process created during system initialization, and never completes its task. Essentially its goal consists of keeping main-store full of processes which wish to use the central processor. The simplified algorithm is: see whether any process wants to be swapped in; swap out processes until there is room; swap in the deserving process; repeat forever. Actually there exists a one-second wait on the clock so that the scheduler, the highest priority process, does not monopolize the processor. A few details of the above algorithm need be examined for later discussions.

The process table 'proc' (recall that the proc-structure, an element in the 'proc-table', contains all the information about a swapped-out process) is searched for the process which has been swapped out for the longest time. If no runnable processes are swapped out, then sched() will wait until some other system routine awakens it, indicating there may be some work to do. When a swapped-out runnable process is found, malloc() is called to determine the availability of free memory. If there is enough to fit the swapped-out process into, then it is simply swapped in, as described later. Otherwise memory must be freed up to accommodate it. First sched() looks to see if there are any

currently resident processes which are waiting for an event, and therefore do not require the resources of the central processor. If any are found they are swapped out and the routine starts over with its search of the process table. If no memory has been found by the two previous methods, the process table is searched again, for the oldest process in memory. If the oldest process has been loaded for at least two seconds, and the process to be swapped in has been on disk for at least three seconds, then the oldest process is swapped out. The scheduling routine then starts over again scanning the process table to find a process to swap in.

Once enough memory has been found and allocated through malloc() the new process is swapped in. First a check is made to determine whether the text portion of the new process is already available in memory. If so, it is linked to; otherwise, it is read into the newly allocated memory. The user area, data area, and stack of the process are swapped in as one chunk, and the space they occupied on disk is freed. The base address of the user-structure (and therefore the whole process) is stored in the proc-structure for the swapped in process, and a flag is set to indicate that the process is loaded. While the process was swapped out, the base-address pointer was used to indicate where on the disk the process was stored. The swapping-out algorithm details are discussed later in this section.

There are a few important points which must be mentioned about the swapping-in routine. First it will be noted that

sched() does not check to see whether the process would fit if the text and the rest of the process were separated. There is no particular reason that the check could not be made, and it would cause very little overhead. Secondly, process priority does not enter into the scheduling algorithm, which makes for a simple algorithm, but doesn't give as much power to the priority mechanisms as it might. Notice also that the scheduler operates independently of the rest of the system (except the clock and the disk routines). Being so independent makes for a clean, easy-to-understand scheduler.

When a process is scheduled for swapping out, all of it is swapped out. The text portion is not written out to disk (its main memory is freed if the current process was the last loaded process using it) as there always exists a good, valid copy of it on disk. The rest of the process is written out to disk in one large chunk. Space is allocated for it by malloc(), before it is written out. The final notion to be emphasized here is that nothing in the process' user-structure is changed. The only items changed are located in the process table and indicate the loaded - unloaded state of the process.

2.7 Departing This World

The termination of a process comes in two asynchronous parts; the death of the child (by invoking the primitive exit(), being killed by a parent, or by some abnormal condi-

tion), and the informing of the parent of the child's death. These two functions are performed by the routines exit() and wait() respectively. The reader will note that these are the routines which implement the system primitives of the same name. First the exit() routine will be discussed, although it does not necessarily have to be the first routine called.

Exit() is either called explicitly or by default, via a return from the top most level of a 'C' program. In either case the execution is the same. The first deed accomplished is to clear all signals which the soon-to-be-defunct process was intending to catch. All open files for the process are then closed, and the text segment of the process unlinked. If the current process was the last process linked to the text its memory on disk as well as in mainstore is released. The volatile portions of memory (the user-structure, data area, and stack) are then freed up, and the user-structure is copied into a disk buffer to be saved for the parent. The process status is set to ZOMBIE, and the address pointer in the process table is set to the address of the disk buffer. Finally the process table is searched and the parent of the deceased, if it has been waiting for a process to die, is informed of the imminent doom of the child. Also any children which the deceased process had created are adopted by the <u>init</u> process (process #2).

When the parent of a child issues a wait() primitive (the <u>init</u> process sits in an infinite loop issuing waits),

the child may finally be removed from the record books (the book is the proc-table). If the parent waits before the death of a child the system will suspend the parent's activity until a child dies. Once a child has died the wait goes into action to tell the parent about it. First the user-structure is read in from the disk buffer where exit() placed it. The CPU, system, and elapsed times for the child are added onto the parent's times (stored in the user-structure). The process table entry for the dead process is set to zero, and the buffer for the user-structure is freed up. Finally the argument to exit() is returned to the parent, as a status indicator.

The important item to note here is that the parent cannot stop its child temporarily. It also cannot examine the child while it is running. The only method of communication between the parent and its children is through pipes. In the next chapter, the MMU will be discussed, to provide a firm base for the understanding of Chapter IV where the concepts explained in the current chapter are used.

Chapter III

The Hardware .

3.1 Introduction

The memory-management unit of the PDP 11/45 provides both access-control and virtual-address mapping, for each memory reference initiated by the central processor. As with most memory-mapping units, addresses generated by direct memory access (DMA) devices are not affected by the mapping device. The Dynamic Address Translation unit of the System 360 Model 67, IBM (1967), is another device of this genre. For a review of several other paging/segmenting machines see Randell (1968). A description of the MMU of the PDP 11/45 is given in Digital (1972), a maintenance manual, and also in Digital (1975). A general characteristic of memory-management devices is their ability to provide a mechanism for executing several different programs referencing the same virtual address, but occupying distinct physical addresses. The facilities provided by the PDP 11/45's memory-management unit include: a separate memory-mapping mechanism for each of three processor modes; access control on a per-segment basis, and statistics-gathering capabilities. These facilities are provided for the system designer but their use is not mandatory. UNIX, for example, does not use the statistics-gathering portion of the unit at all. A variety of systems may be constructed using the memory-management hardware, from the most primitive memory-

resident single-user system, to an elaborate paging multi-user kernel-domain system, Spier (1973), and Spier (1973a). UNIX lies somewhere between these two extremes.

The first section of the chapter discusses which of the six different sets of mapping/access-control information structures will be selected for a particular memory reference. In the second section, the algorithm for the formation of a physical address from a virtual address and an information structure, will be explained. In the fourth section the various levels of access-control and statistics-gathering information will be explained. Throughout the chapter, examples from UNIX will show how it currently uses various aspects of the hardware.

3.2 Processor Modes

The parameters (relocation and access-control information) used for any particular address translation are selected from six sets of parameters by three items: the current mode of the processor (kernel, supervisor, or user); the data space enable bit; and the type of reference (instruction or data). The various modes in which the central processor is capable of executing programs will be discussed first.

The central processor of a PDP 11/45 is capable of executing instructions in three different modes: kernel, supervisor, and user. The current and previous modes are retained in the program status word. To differentiate between

three modes requires two bits of information, which are capable of four values. The one-zero combination is not used and causes a trap if a program ever generates it. The reason for this somewhat strange situation (i.e. why the one-zero combination?) is to provide some protection. These bits in the program status word are capable of being cleared only by loading a new program status word (and program counter) from a trap vector. Therefore control may only be passed outward to less privileged modes, by setting these bits; from kernel-mode (00), to supervisor (01) and to the least privileged user-mode (11).

When in kernel-mode the machine is capable of executing certain privileged or critical instructions. These instructions would cause problems in system integrity if their usage were undisciplined. A few examples are: set priority level of the processor; 'halt' the processor; 'wait' for an I/O interrupt; and 'reset' all input/output devices. One other feature of the kernel-mode which makes it more privileged, is the 'vectoring' of all processor and input/output traps through the kernel data memory-management set. That is, the new program counter and program status word are contained in a two-word vector relocated through the kernel data space. The new program status word may, of course, have its current mode set to either kernel, supervisor, or user mode. Because of this the kernel-mode program has control over all service routines but may pass it on to the other modes if the situation warrants it. Since the

kernel-mode is given these privileges by the hardware it is normally used to implement the lowest level of an operating system, commonly referred to as the 'kernel', Spooner (1971). UNIX uses exclusively the kernel-mode to implement all input/output drivers, clock watchers, trap handlers, and all of the system primitives. As UNIX is written in a high level language, 'C', which is not capable of producing privileged instructions, many of the privileges provided by the hardware are regulated. An analogy may be drawn to the Burroughs B5700 system, Organick (1973). The B5700 system uses software protection since no hardware protection exists on the Burroughs machine.

The supervisor and user modes of the central processor are the same as far as hardware privileges are concerned. The only hardware difference between them is which mapping-register set is used, and which stack pointer is to be used. Each of the three modes has its own stack pointer (register six) and the current processor mode specifies which one to use. Just because there are insignificant hardware differences between user and supervisor modes does not imply that they must perform similar functions. In fact, in a sophisticated environment the supervisor-mode may be used to implement high level input/output, common routines, or real time processes. UNIX avoids the issue completely by never using the supervisor-mode, as UNIX was intended to be capable of being run on PDP 11/40's, which do not have that mode. Thus, in UNIX the kernel-mode address

space contains all the code for the system primitives, as well as the sensitive data structures. The user-mode is used for the execution of user processes and is remapped each time a new process is scheduled for execution.

The second factor used in determining which of the six sets of relocation and access-control information to select for a particular memory reference, is the knowledge of the type of reference; i.e., is it a data or instruction reference? The use of the separation of data and instructions is optional and must be enabled by setting a bit in a status register. It may be enabled for any combination of processor modes (kernel, supervisor, and/or user) by setting one bit on for each mode to be enabled. The current version of UNIX always leaves this feature disabled, but there are rumors that Version 6 has separate instruction and data spaces in the kernel-mode, Perentz (1975).* One might wonder about the advantages of enabling the separate data space. The main motivation is space; a pure procedure need occupy only the instruction space while pure data (that which is not executed) need only occupy the data space. When the data and instruction spaces are separate (that is non-overlapping for the most part) a larger program may be accommodated in the virtual address space. In theory, the program may be up to twice as large, but it is rarely achieved in the real world. The second possible motivation for separating the instruction and data spaces is to provide execute-only protection.

* The rumor turns out to have been correct.

If the pure code of a proprietary program resides only in the instruction space, then it may be executed but cannot be read or written by a user program.

Any memory reference which causes the program counter (register 7) to be incremented (by two) during the fetching of a mainstore word, is considered an instruction fetch and is performed through one of the 'I' space register sets. All other mainstore references are mapped through one of the 'D' space register sets. The set selected is determined by the current mode (user, supervisor, or kernel) of the processor, as explained previously. The aforementioned rule is very general, and includes all cases of fetches performed through the 'I' space, but doesn't give a very good feel for when the 'I' space or 'D' space is used. To give a better understanding of the difference a description of the various memory references will be given. For each instruction executed the instruction itself must be fetched from mainstore, which increments the program counter and is therefore fetched through the 'I' space. If the instruction requires any operands they must also be located. Whether these operands are to be accessed through the 'I' space or 'D' space depends on the addressing mode bits in the instruction, and the register specified. For an explanation of the format of PDP 11 instructions see Chapter III of the 'PDP 11/45 Processor Handbook', Digital (1975).

Register mode (mode 0) accesses the operands directly in the hardware registers and therefore uses neither the 'I'

space or the 'D' space. Mode 1 (register deferred) uses the value in the register designated to access the operand in the 'D' space. Auto increment mode (mode 2) is capable of using either 'D' space or 'I' space. The former is used if the register specified is zero through six, but the latter is used if the register specified is seven, since it is the program counter, and is incremented. When the program counter is used with mode two the construction is called 'immediate' since the data used immediately follows the instruction. Auto increment deferred (mode 3) is similar to mode 2, in that if registers zero through six are specified, the indirect address is fetched from the 'D' space, while if register seven is specified the address is fetched from the 'I' space. The data in auto-increment-deferred mode is always accessed through the 'D' space memory management parameters. Auto decrement (mode 4) and auto decrement deferred (mode 5) do not increment the program counter and therefore are all accomplished through the data space. The use of register seven in either of these constructions is not encouraged, as disaster may result. The final two modes operate quite similarly, indexed (mode 6) and indexed deferred (mode 7); both obtain the index via a direct reference to the program counter which is then incremented. Therefore the index is obtained from the instruction space, but the indirect address (only in mode seven) and the data are mapped through the 'D' space.

There are two other types of memory references which are

performed on PDP 11's: input/output direct memory access references, and traps. As mentioned before, traps are mapped through the kernel data space. If the data space is not enabled the mapping is performed through the instruction space. The other type of memory reference is done by I/O devices which are capable of transferring information directly to and from mainstore, without processor intervention. The key word here is 'direct'; that is, these memory references do not pass through the MMU, but are real physical memory transfers. Therefore the system primitives which initiate the input/output operation must place the physical address of the buffer in the device's bus-address register. In UNIX this is quite easily accomplished because the kernel address space, where the I/O buffers reside, is mapped into physical memory on a one-to-one basis.

To summarize the above section, one of six different sets of relocation and access control information parameters is selected by the mode of the processor (kernel, supervisor, or user) and the type of the reference (instruction or data). Each set of parameters contains eight double-word pairs which describe the relocation and access allowed for a particular page of storage. The next section describes the mechanism used to select the relocation information, and how it is used to form a real or physical address.

3.3 Relocation

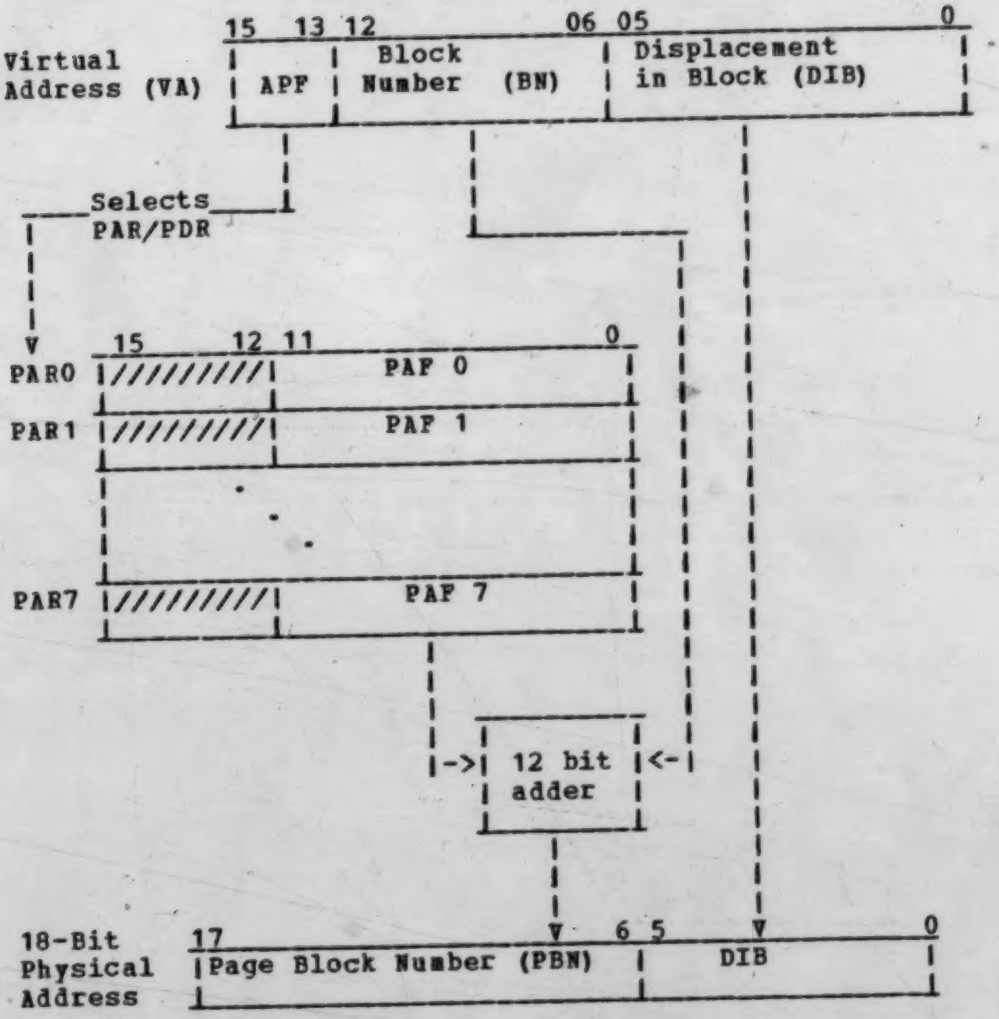
When a memory reference is generated by the processor

the memory management unit first selects one of six sets of relocation and access-control parameters. The selection mechanism has been described in the previous section. These sets each contain eight double-word pairs which describe eight memory pages. The words of the pairs are called the page address register (PAR) and the page descriptor register (PDR). The relocation information is contained in the PAR, while the PDR contains the access control and statistics information. This section will describe the selection and use of the PAR to generate a real address from a virtual address. Even though the PDR is selected via the same method as the PAR and they are used as a pair, the PDR and its functions will be described in the next section, as it differs considerably from the PAR.

As mentioned above each set of relocating registers contains eight PAR's and eight PDR's. The current virtual address is used to select which one of the eight PAR's will be used for any particular address translation. The three high order bits, referred to as the active page field, are used to select one of the eight possible PARs. Therefore if the three high order bits are 101, for example, the fifth PAR will be used. Each of these eight registers contains the relocation information for one 'page' (4K-words) of the virtual-address space. This implies that a program may have at most eight data and eight instruction pages, capable of addressing 32K words total. The page address register contains the base address of the page in real physical memory.

The base address is twelve bits long and is positioned in the lower twelve bits of the PAR. Figure 4 shows the construction of a physical address from a virtual address. In brief, the process is as follows: the PAR is selected by the active page field of the virtual address; the block number (bits 6-12) of the virtual address are added to the contents of the selected PAR, and the resulting sum is concatenated to the displacement in the block (the low order six bits) of the virtual address. The result of the algorithm is an eighteen bit physical address.

There are a few implications of the mechanism which are not apparent at a cursory glance. Most important is the justification for adding the base (value of PAR) to the block number. In other machines, for example the IBM 360/67, IBM (1967), the base is simply appended to the displacement, saving 32 nanoseconds, the time required for the addition. The advantages of the method used on the PDP 11/45 are that pages do not have to start on physical page boundaries; they need start only on 32-word block boundaries. This represents a considerable saving in fragmentation of memory, as pages are quite large (4096 words) compared to blocks (32 words). A second feature of the scheme is that the relocation registers are kept in high speed storage (physically in the memory management unit), which reduces the translation time to 90 nanoseconds per reference. Because of the short translation time, more exotic pieces of hardware, such as associative arrays, are not required to maintain the speed



Construction of an 18-bit
Physical Address

Figure 4

at some acceptable value. Finally, these registers exist in the normal physical-address space of the machine, and therefore may be protected from user programs by not including them in the address spaces of supervisor and user modes. Because they occupy the same physical page as the input/output device registers they are so protected.

3.4 Page Descriptor Registers (PDR)

In parallel with the relocation described in the last section, access-control information is checked and statistics are accumulated for the accessed page. The information and the statistics are maintained in the PDRS. As with the PARs, there are eight PDR's, one of which is selected by the active page field in the virtual address. Each of these PDR's contains statistics and access-control information. As these items are separable they will be described in separate sections. First the access-control information will be described.

3.4.1 Access Control

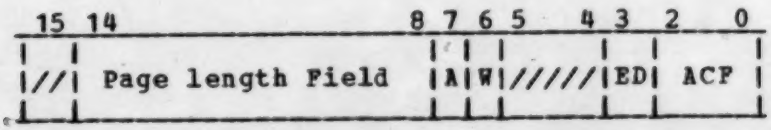
The main function of the PDR is to provide page-length, page-expansion, and access-control information. Pages are of variable length and may have from one to 128 blocks of 32 words in them. This implies that the maximum size of a page, and the size it must be for a continuous virtual-address space, is 32 x 128 = 4096 words. The current allowable size of a page is contained in the page-length field of the PDP. By convention, stacks on the PDP11 expand to-

ward lower addresses; therefore, the designers of the MMU provided the capability for downward expansion as well as the normal upward expansion of data areas. The capability is contained in the expand downward bit; if it is turned on, then the page-length field is the lowest block number which is a valid address. To summarize the page length field usage, a quote from the "PDP 11/45 Processor Handbook" is provided:

> "A Length Error occurs when the Block Number (BN) of the virtual address (VA) is greater than the Page Length Field (PLF), if the page expands upwards, or if the page expands downwards, when the BN is less than the PLP."

When a length error occurs, the current instruction is aborted, and a trap is made to the MMU trap vector.

The second piece of information in the PDF determines what type of accesses are allowed in this page. The three major types of access are: no access (page not resident, or nonexistent), read only access (abort any write attempts), and full access (page may be both written and read). A fourth access type exists through the use of the 'I' space, namely execute only access. Three minor quantifiers are added to these basic modes to aid in gathering statistics of page usage. The various combinations are listed in Figure 5. The difference between an abort and a trap is that the instruction is completed on a trap condition, but is stopped, perhaps partially completed, on an abort. Thus traps are



Page Length Field - The number of 32-word chunks which the selected page is allowed to access. It is compared to the Block Number of the current virtual address.

A - Set when an access to this page met the trap condition specified in the Access Control Field (ACF).

W - Set if this page has been written into since the PDF/-PDR was last written into.

ED - Expansion Direction; ED=0 => expand upward; ED=1 => expand downward.

ACF - Access Control Field; possible values and their definitions follow:

Value	Description	Function
000	non-resident	abort all accesses
001	read-only and trap	abort on write attempt; memory-management trap on read
010	read-only	abort on write attempt
011	unused	abort all accesses-reserved for future use
100	read/write and trap	memory-management trap upon completion of a read or write
101	read/write and trap on write	memory-management trap upon completion of a write
110	read/write	no system trap/abort action
111	unused	abort all accesses-reserved for future use

Page Descriptor Register (PDR)

Figure 5

used for statistics gathering, and aborts hinder disalloved accesses.

3.4.2 Statistics Information

In each PDR there are two bits which log usage information about the page described. The 'W' or written-into bit is always active, and will advise the system, if interrogated, whether the page has been altered since the page descriptor register was last updated. The 'W' bit is a read-only bit (as is the 'A' bit discussed later) which is set whenever the page is altered, and is cleared whenever the page address or page descriptor register is updated. In disk-swapping applications the 'W' bit can be used to determine whether a page marked for removal from mainstore (by some paging algorithm) needs to be rewritten to disk, or that the copy on disk and in memory are the same, implying that the copy in mainstore may simply be overwritten.

A second bit of information provided by the memory-management unit to aid in paging applications is the attention or 'A' bit. This read-only bit is set whenever any of the memory-management trap conditions is true because of a particular page access. These various trap conditions are specified in the access-control field described in Figure 5. The option of whether the trap is taken is independent of the setting of the bit in the PDR, since the trap must be enabled by a bit in status register 0. Thus the 'A' bit can be used as a 'has this page been referenced' indicator

without the overhead of the trap. The 'A' bit gives the systems designer much latitude in the paging algorithm used, by allowing him to interrogate the bit with various strategies. Once the 'W' and 'A' bits have been recorded, the simplest way to clear them is to write the PDR back into itself.

UNIX currently doesn't use either of these bits; in fact, it blindly writes all pages to be swapped out to the disk. Because the data portion of a process does not exist on the swapping disk, while the process is in main store, all of the pages must be written when the process is swapped out. This makes for a simple, although somewhat inefficient algorithm. The next chapter outlines the steps required to improve UNIX's usage of the quite powerful memory-management unit.

Chapter IV

The New Software

4.1 Introduction

A number of experiments which could be performed on UNIX to restructure it into what is conventionally termed a paging system are described here. The motivation for the extension is that the hardware is capable of doing more work than it currently does. The term hardware here means of course the MMU. The additional work refers to the capabilities of the MMU which are not currently being exploited. These include separation of instruction and data spaces; using the statistics information which the unit provides; and loading programs into non-contiguous physical memory.

The chapter is the crux of the thesis, and discusses some of the possible experiments which could be performed on UNIX. The restructuring of UNIX to a paging system can be broken down into four independent phases or stages. The chapter is subdivided into sections representing these stages. In each section the advantages of performing the described modifications will be given, as well as detailed descriptions of what must be changed and why it must. To understand the intricacies of these changes, a detailed knowledge of the operation of UNIX, as well as a complete knowledge of the hardware of the MMU are required. The two preceding chapters have provided the necessary background for and understanding of the hardware capabilities, and were

necessary because no other adequate descriptions exist.

Although the four stages of development proposed must be completed in order; they are independent to the extent that when each stage has been completed a working system will exist. In brief the sections to be covered are: separate the instruction and data spaces; provide for discontinuous physical mainstore images of user processes; keep an image of user process on disk to reduce rewrite time; and finally bring in user segments only as they are needed (allowing a process to run without being totally loaded). These four stages are of increasing difficulty to implement, and the fourth stage requires changes to so many systems routines it is doubtful that implementation is warranted. The order in which these are listed above and described below is the easiest method of implementing them; in fact, the last two stages are very dependent on the previous stages being implemented.

4.2 Stage I Separate Instruction and Data Spaces

Perhaps the most obvious portion of the MMU which UNIX neglects to utilize is the separation of the instruction and data spaces. The main advantage of having the instruction and data spaces separate is that programs may be up to twice as large as they could be without the separation and with very little work on the part of the system programmer. The actual routines which need to be rewritten or modified and the structures which must be modified to accommodate the

change are described in detail later in the section.

Although the address space of almost any computer could be separated into independent instruction and data spaces, this was not done on most previous machines as the advantages didn't outweigh the disadvantages. The turning of the tide came on the PDP11 series computers when the need for a larger mainstore than could be addressed with sixteen bits arose. The problem is more pronounced on the PDP11 than on most other sixteen-bit machines because the PDP11 is byte-addressable, effectively leaving it only fifteen bits to address memory. This limits the size of programs and data to 32K words (2 bytes/word). As the designers of the PDP11 series felt 32K was a severe limitation they built hardware to separate the instruction and data spaces of their top-of-the-line computers, the PDP11/45's and PDP11/70's. Besides providing up to twice the size of program that can be accommodated, the separation also provides a new type of protection, execute-only, which means that not only can the executable portion of a program be protected from being written into or read from, but also that genuine data cannot be executed, either by accident or malice. Despite the fact that the hardware is capable of distinguishing between instructions and data, UNIX makes no use of this mechanism. UNIX is made to run on hardware-limited PDP11/40's as well as 45's and 70's. The PDP11/40 series does not have the separate instruction and data space feature.

The advantages of a separate instruction and data space

are, in brief: potentially it can double the size of a program which can be run; it provides execute protection (and non-execute protection). The overhead induced (or disadvantages of using the feature) is relatively small. Basically the overhead is in a small amount of extra storage required (16 words) in each process' user-structure, and the slightly longer amount of central processor time required to switch processes. The components in UNIX which need to be changed are the kernel, linkeditor (and associated files e.g. size(I) and nm(I)) and the debuggers. The kernel needs to be changed for the obvious reason that it is the heart of the matter. The changes to be made in the kernel will be explained later in complete detail.

The only system structure which needs to be modified to accomplish separation of user-instruction and data spaces is the user-structure. It contains space for the prototype memory-management registers. There is exactly one user-structure for each process in the system. As previously mentioned the structure is swapped out with the rest of a process, when a swap is made. Therefore an extra sixteen words added to it will not be a very critical increase. (If the structure were always resident then the increase might become critical in some applications). As the user-structure does not reside in the user's address space (see Figure 2 of Chapter II) the clientele who do not use the separate address spaces cannot claim that the 'feature' is causing them unwarranted injustice. The new space in the user-structure

will be used for the memory management prototypes, eight words for the instruction space PAR's and eight words for the PDR's. How these words are filled and used is described in the next paragraph.

The two main routines which manipulate the memory management register prototypes are: estabur() (establish user registers) and sureg() (switch user registers). Basically estabur() calculates the offsets required by the MMU to allow the user to address his text, data and stack. Sureg() then adds in the base address (recall that the program is contiguous so only one base is required) of the current location of the process and places the results in the actual memory management registers. These two routines must be changed to allow for separate instruction and data spaces as there are sixteen elements to fill instead of eight. Sureg() is quite easy to change since only the number of registers loaded must be increased from eight to sixteen. It can of course be assumed that the sixteen prototype elements have been properly set up for this routine.

Estabur() requires a few changes to accommodate separate instruction and data spaces. First a parameter should be added to indicate the user's desire to separate the instruction space. Because two routines call estabur() during the execution of a process, as well as exec() during the initialization of the memory management registers, the separation parameter must reside in the more permanent user-structure. These two routines, sbreak() (the system primi-

tive to expand and contract data areas) and trap() (when it expands the stack) have no other method of determining whether the instruction and data spaces are separate. Exec() can find out by investigating the 'magic' number in word zero of an executable file and should be modified to change u_sep according to its value. The magic number could have the value 0411 (currently the loader ld(I) places 0407 for normal files and 0410 for reentrant read-only text files in the magic number). Adding a new parameter to the loader would enable the user to specify that he wants to separate instructions. Because a new 'magic' number is to be added (as well as relocating the data to start at address zero), several utilities must be changed to recognize it. Among them are the debuggers, size(I), and nm(I) (an external name locator).

The first item in estabur() to be changed is the method of checking to see whether the new (larger?) process will still fit in memory. The algorithm must be changed to reflect the value of the u_sep flag. The text-filling algorithm need not be changed, but if instruction and data spaces are separated the unused instruction space registers should be filled with zero, permitting no access. This filling will conveniently move the pointers so that when the data space is filled next, it will be placed in the proper array elements. If the data and instructions are not to be separated, the final action of estabur() will be to copy the instruction-register prototypes into the data-register pro-

totypes. As can be seen, from the above detailed changes, a well-written operating system is pleasantly simple to modify.

Besides these two routines which deal directly with the segmentation registers there are a few incidental details to be taken care of before instruction and data spaces can be separated. Of course, one of these is that the separation must be turned on (by setting the memory-management status register to four) during the initialization of UNIX. It can be done either in the assembly language start-up routine start(), or in the first 'C' language routine entered at initialization time, main(). The other change which is required concerns the trap handler for system calls. Trap() must be capable of fetching data items from both the instruction space and the data space. Following is an example of the calling procedure for a system primitive:

```
_times: .txt                    /place following words in I space
        mov   2(sp),A           /store user parameter in list
        sys   indr;B            /issue indirect system primitive,
                                /pointing to real system call
        rts   pc                /return
        .data                   /force to D space
B:      sys   times             /real system call, interpreted
                                /by trap()
A:      ..                      /parameter
```

In the above example the address of 'B' must be fetched from the instruction space, while the actual system call and its argument must be fetched from the data space. To facilitate this (there is a simple, well-defined algorithm) a new assembly language subroutine must be added to fetch words

from the instruction space. Trap() will then have to be rewritten to implement the required algorithm. Fundamentally the algorithm proceeds as follows: if the program counter must be incremented to skip over the data item then fetch the data from the instruction space, and otherwise from the data space.

This stage of experimentation with memory segmentation in UNIX has been described in detail to give the reader some notion of the simplicity of changing a well-structured operating system. The other three stages will not be described in as much detail to avoid needlessly confusing the reader.

4.3 Stage II Discontinuity of User Processes

Chapter II described the relatively lengthy process of expanding a user process. The main cause of the inefficiency is the simple fact that a user process must occupy a single, continuous memory area. A method to alleviate this and other problems, derived from the designers' principle of forcing processes to be contiguous in physical memory, not to be confused with the normally-fragmented user address space, is explained here.

The advantages of allowing processes to occupy non-contiguous mainstore (that is, to have mainstore managed on a segment basis) are actually twofold; first, it provides an easier method of performing process expansions, and secondly more processes can fit into a fragmented memory. The first

advantage allows the process expansion algorithm to be rewritten in a much more efficient manner. The new algorithm would be something like the following:

1) allocate the new space (it must be broken up into segments merging the new data (or stack) with the old one) by calling malloc().

2) copy the old partially full segment into a new larger one.

3) clear the rest of the new space.

4) free up the partial segment.

The new algorithm guarantees that the largest piece which needs to be copied is less than 4K words, as opposed to the whole process. The second benefit attained by managing mainstore on a segment rather than a process basis is that a larger number of processes may be simultaneously present in memory. Because each allocation of mainstore is at most 8K bytes, (the length of the longest possible segment) as opposed to a complete process (which normally occupies more than one segment) more processes should fit. As the size of the pieces to be allocated is smaller, the likelihood of filling up a hole is better. The reader should also recall that the purpose of these changes is to convert UNIX into a paging operating system, and one of the steps which must be implemented is the restructuring of UNIX to manipulate discontinuous user processes.

Unfortunately this stage of implementation besides having some very strong points in its favor also has a few

negative aspects. The main problem has to do with input/output. As normal input/output is done by copying the data to and from kernel buffers (using MFPD/MTPD) there is no problem with it. The buffers reside in fixed locations in the kernel and therefore each buffer is a continuous portion of physical memory. The problem exists only in the case of physical input/output. This mode (as implemented for all block type devices: disks, tapes, etc.) uses the user's physical mainstore as the source/destination region. That is, the physical input/output routine (physio()) simulates the MMU (recall that in the 11/45 series the input/output does not go through the memory management unit), and then sets up the device to do the input/output to the user's physical memory. Because the devices on the PDP11 are not capable (because of hardware design) of performing data-chaining operations, physical input/output cannot occur across non-contiguous segment (buffer) boundaries. Therefore the physio() routine must check for them, and take some alternative action (for example, allocate a contiguous dynamic buffer large enough and then copy data). Of course a simple solution to the problem would be to generate an error when it arises. The only other negative aspect of stage two is the slightly higher overhead in managing and manipulating the larger number of mainstore chunks. The overhead is quite small, compared to the present manipulation.

The changes required to implement step two apply to only a few system routines. None of the system support (utility)

routines need to be changed. The routines in the UNIX kernel which must be modified are those most closely connected to the memory management unit; sureg(), estabur(), sched(), malloc(), xswap(), xfree(), expand(), xalloc(), and xccdec(). Although there seem to be a large number of routines and therefore a sizeable task, it is not as bad as it looks. The task may be broken down into four subtasks: swapping in, register copying (to give a process control of the CPU), allocation (deallocation) primitives and allocation (deallocation) support routines. The most basic of these subtasks is the mainstore allocation/deallocation problem, so it will be discussed first.

As UNIX is currently written, all the mainstore for a process is contiguous and therefore is obtained from the system pool in one chunk. This stage of implementing a segmenting system in UNIX decrees that the space for a process must be handled on a per segment basis, which implies that estabur() cannot exist in its current state. Estabur() performs two functions: it checks to see if a process will fit within the constraints of the hardware (eight data and/or text segments and within the physical main store bounds) and also fills up the prototype segmentation registers (as if allocated memory were contiguous). Estabur() should be split up into its two functions and the function of filling the memory-management register prototype should be left to the memory-allocating routines. Therefore there should be a new routine written to allocate memory for a process (on a seg-

ment basis) and fill the prototypes at the same time. In the same respect, the memory-management prototypes should be adjusted to reflect memory when it is freed. The malloc()/mfree() routines are used by other routines in the system, and must therefore remain unchanged but the data structures which they work on (coremap) must be accessed by the new routine.

The swapping-in process, as performed by sched(), must be modified to reflect the notion of noncontinuity. Currently, sched() checks only to see if there is a chunk of mainstore large enough for the whole process. Obviously sched() must be modified to find the pieces of main store which the process requires. As the memory-management prototypes exist in the user-structure which is (wisely) swapped out with the process, sched() has no method of determining whether the fragments of memory which exist are capable of holding a process (the entire process must be resident for it to be executable - in stage four this restriction is lifted). To aid the scheduler (of mainstore) therefore, a new data structure must be maintained. This structure (either pointed to or appended to the proc-structure) would include a byte containing the current length of each of the sixteen possible segments. This small (eight word) structure is all that is required for a new scheduling algorithm to be written. Of course, many algorithms could be constructed to pack as much as possible into the available space. The new data structure will be maintained by the allocation/deallo-

cation routines.

The allocating/deallocating support (xswap(), xfree(), ...) must be changed to reflect the modifications performed on the allocating routine. The checking routine will have to be called to see if the proposed (by the user) expansion will fit. Then the actual memory will have to be allocated, and the memory management register prototypes filled. Because of the new structure of a process' mainstore image, (it will still be contiguous on disk when swapped out), the amount of information which needs to be moved around in storage as a process expands is minimal.

The final routine which requires modification is sureg(), which copies the prototype registers into the real hardware registers. Currently, the base address of the process is added into the registers as they are copied. As there are many bases (one for each segment) and the real addresses are contained in the prototypes, the addition is not required. The prototypes may simply be copied to the real registers.

Provision for noncontiguous processes is a large step, and deviates from the intended structure of UNIX. It is because of this deviation that the changes are so involved. Providing for noncontiguous processes is a major change in the structure of the system and is difficult to perform because of it. As will be seen in the next stage, changes which are planned (or anticipated) are more easily imple-

mented.

4.4 Stage III Paging Processes

Stage three of experimentation on UNIX's memory management structure brings UNIX one large step closer to what is classically termed a paging system. There are three interrelated aspects of the changes required to accomplish step three. First, a copy of each existing process will be kept on the swapping disk; second, the process' image will not necessarily be contiguous (it will be broken up into segments as mainstore is); and third, when a writeable segment is to be swapped out the written-into bit will be examined to determine whether the mainstore copy is the same as the one on disk. The last feature does for data areas what 'text' segments did for instructions. One item which should be noted here is that a process will still be completely loaded in mainstore before it is allowed to run. Removal of the final restriction is discussed in the next section.

The inherent advantage of this step is quite obvious - it cuts down the amount of data traffic on the swapping medium. If the swapping disk is also used for other data items (for example, on our system the temporary files share the swapping disk), it will lessen disk access contention. Also the smaller number of data bytes transferred will reduce the traffic on the UNIBUS, and therefore speed up mainstore operations (for example instruction and data fetches). Besides these advantages there are a few disad-

vantages. Additional information about each process will have to be kept and manipulated in mainstore. The exact nature of the information will be described later. The fact to note here is that the data will occupy mainstore (there can be a total of only 128K words on a 11/45) and will have to be constantly manipulated by the system, thereby stealing valuable CPU resources. The real question lies in whether the advantages outweigh the disadvantages. Unfortunately there is no practical way of determining the answer without actual implementation and benchmark tests (quite a sizeable task).

The additional data structures required to implement stage three will contain: 1) the composite written-into bits, and 2) the swapping disk addresses of the sixteen possible segments. The written-into bits (one for each of the possible data segments, eight total) designate whether the mainstore copy of a segment is identical to the one on disk, and therefore whether it needs to be written out on a swap. The second structure (thirty-two words long, two for each segment) contains pointers to the areas on disk which are reserved for each segment. As the methods of manipulating the second structure are inherently simpler they will be discussed first.

The sixteen disk-area pointers remain constant as long as the size of the process does not change. Only when a process expands (through automatic stack expansion, the break() primitive, or by the exec() primitive) or contracts (via

the break(), exit(), or exec() primitives) need these pointers be altered. As most disks are capable of being addressed in 512-byte chunks (sectors), the pointers need only address these chunks. These are the same sized information structures as are currently maintained, in 'swapmap', by malloc() and mfree(). These routines may still be used to maintain the available space on the swapping disk. An indicator must be made to show that a particular segment is empty (doesn't exist on disk). It could be done by filling that slot in the pointer data structure with minus one. The length of a segment (they can be from 64 bytes to 8K bytes long) could be determined from the text, data, and stack sizes and the separation flag, but for efficiency's sake a separate sixteen words should contain the length. When a process is swapped out, the length-of-segment array can be swapped out with it, if there is some way to get it in. A method already exists to bring in the user-structure; - a pointer to its disk address is in the permanently-resident proc-structure. Therefore if the user-structure were swapped in first, along with the length-of-segment array, the whole process could then be found and swapped in.

The second data item required for stage three, although shorter, is much harder to manipulate. As the written-into bit in the PDR cannot be written (see Chapter III), it must be saved, in a composite manner, each time the PAR's and/or PDR's are to be written into. Not only are the PAR's and PDR's written each time the CPU is scheduled to another user

(through swtch()), but also during process expansion, and even the device 'mem' (which treats physical memory as an input/output device). There are even more subtle problems: when input is done from a block device (e.g. disk or magnetic tape) the information received is copied from kernel buffers to the user's buffer with a routine called copyseg(). Copyseg() copies information around using the supervisor-mode segmentation registers. (The only place in UNIX where they are used except clearseg()). Because the supervisor-mode registers are used the written-into bit in the user's PDR doesn't automatically get set. Physical input/output is another culprit: it does input/output directly to/from the user's memory without the MMU. Therefore the written-into bit would have to be simulated in physio(). All of the small kernel routines which move single words or bytes into the user space use the MTPD (move-to-previous-data) instruction, which properly sets the written-into bit.

The major routine which is involved in the swapping is sched(), which would require modification. Recall, in section 4.3 sched() was modified to read in (and write out) noncontiguous processes. It is a relatively simple task to modify sched() to check the composite written-into bit for each segment before writing it out. A new swapping-out algorithm could be developed to swap out those processes which have modified the least number of segments first (of course, the current time-based algorithm would still be the major

factor in swapping a process out). A new scheduler could be developed which would continuously swap out segments which have been written into. This is the method used in some operating systems, including the Michigan Terminal System on the IBM 360/67 (370/168,...) Alexander (1972). These look-ahead writes would probably degrade the performance of the UNIBUS and memory enough that it would not be advisable to re-orient the system to them. The simpler demand scheme currently used would be much more suitable on PDP 11's. The demand scheme stops the process, and then swaps it out (or ensures that the disk and mainstore copies are the same).

Newproc() and exec() are two routines which would have to be modified in a similar manner to implement paging. These two routines both expand (and in the case of exec(), contract) the size of a process. When the size change (or in the case of newproc(), copy of a process) is made the disk space required must be allocated (deallocated). A new subroutine is required to manipulate the disk space, and copy to it.

The final major element which needs changing is the expand() routine. It is called by trap() (to expand the stack) and the break() primitive (to expand or contract the data area). Expand() encounters the same problems (and solutions) as exec() and newproc() which were previously discussed.

When stage three has been completed, processes will still have to be fully loaded to run; the next section deals

with this restriction.

4.5 Stage IV Paging Segments

Stage four examines the final problem in making UNIX a paging (segmenting) system; that of not requiring a process to be completely loaded before enabling it to run. Modifying UNIX to be a paging system is what the chapter (and the thesis) set out to accomplish. Before jumping into the actual implementation some of its advantages and disadvantages will be investigated.

The prime motivation in implementing stage four may be academic in nature since it provides very few advantages to the user, and has many aspects which are disadvantages to both the user and the system. When the modification is examined from an academic viewpoint there are many benefits to be gained by its implementation. As UNIX is a well structured operating system written in a high level language it is both beneficial to students and easy for students to examine the elements which make it work. If stage four were implemented it would provide another element to be examined and studied. Also, the more sophisticated students would be provided with a base to test various segment replacement algorithms. The UNIX system runs on a machine which is cheap enough that almost any university can afford to let the student have (on a temporary basis) his own system on which to modify and test various ideas. The actual (non-academic) benefits gained by implementing the fourth step are as fol-

lows. Most important is the fact that if a process doesn't have to be completely loaded to run, larger processes may be run than would otherwise fit on the machine (limited of course by the address space of the machine). These days this is not a strong point as memory prices are rapidly falling. To buy enough memory to accommodate the largest possible process (32 K bytes of instruction and 32 K bytes of data) costs around five thousand dollars, or approximately five to ten percent of the cost of the whole system. The second advantage is that more working sets of processes can fit in mainstore than can entire processes. For a discussion of working sets see Denning (1968). Keeping only the working sets in main memory would lessen the number of paging operations required to run all of the available processes. The effectivness would have to be determined experimentaly. One item to note here is that a single PDP11 instruction may reference up to six segments, two in the instruction space and four in the data space. An example follows (the addresses are absolute and are for reference only):

```
017776        mov  @src,@dst /crosses segment boundary
030000 src:   s1             /indirect address of source
040000 dst:   d1             /indirect address of
                              destination
050000 s1:    x              /actual source
060000 d1:    y              /actual destination
```

Even though the above example is contrived it shows that the working set of a PDP11 program may be a large percentage of the sixteen possible segments. Another example of this

phenomenon is the IBM 360/67, which can access eight pages with one instruction. The final argument in favor of implementing step four is that it reduces the amount of input/output done to the paging disk. A properly running paging algorithm would not bring segments into memory which were not subsequently used.

The major disadvantage of implementing the fourth stage is the increased overhead associated with the bookkeeping required to determine the working set of a process. Not only does the system have to establish use-counts for the segments, but also there is increased overhead in having to switch processes frequently, because the current process requires a new segment to be swapped in. Another added system overhead is in determining whether a trapped reference to a segment is caused by a reference to non-existent memory, or because that segment simply is not loaded. This is especially sticky with the stack segment(s) as they allow automatic expansion. The final disadvantage of implementing step four is the added complexity and size of the system. As the kernel is always resident and uses only the kernel-mode memory-management registers it is limited to 32K bytes of data or instructions. This may become a limiting factor in the complexity of algorithms which can be implemented. An interesting experiment would be a conversion to a two-level supervisor using the supervisor-mode of the PDP11/45. As mainstore is quite cheap the second level could remain resident in the manner of the kernel. As it would require

an almost complete rewriting of the system (the principles and functions it performs could remain the same) it will not be discussed further here.

The actual implementation of stage four affects quite a few routines (in a consistent manner) and therefore will be discussed in principle rather than in detail. Currently there are three types of page faults: stack expansion faults, and two types of non-existent memory faults, the two types are handled differently. The first type is 'expected' by a machine-language system routine, that is, it will only cause an error indicator to be sent to the originating process, rather than sending a signal. Expected segment faults are normally associated with the system performing some service for the user in the user's memory; for example copying system input/output buffers to or from the user's data area. The second type of segment fault is normally concerned with a bug in a user's program (for example, array subscript bounds exceeded or incorrect pointer used) and causes a signal (which can be trapped or ignored by the user) to be sent to the user process. In all three of these cases (that is, each time a memory management trap is caused), the first fact which should be determined is whether a segment needs to be swapped in. If it does, then the swapper should be called to start the swap and then the CPU resource should be switched to a process which can use it.

The implications of this simple procedure are very subtle and cause many problems with the nature of system

routines. One major implication is that the CPU may be rescheduled without an explicit call to swtch(). Two examples of problems lie in the machine-language support. If the system was prepared to catch the fault (for example in fubyte()), the priority of the CPU will have been set to seven (non-interruptible) to alleviate the locking problem (it is in a critical section). Therefore, when the process is restarted, 'nofault' (the variable which indicates a routine is prepared to catch genuine non-existent memory faults) and the processor priority must be re-established properly; currently, it is not done since other faults never happen. The major portion of the change would be in the swtch() routine. Another portion is the copyseg() routine. This system subroutine (used to copy around expanding segments, etc.) temporarily changes the previous mode of the CPU to system-mode. If a page-missing fault occurred here the old program status must be saved and restored properly.

Another major problem (implying that a routine or routines need to be modified) has to do with forking. Recall that newproc() (section 2.2) expected the process which called it to be completely loaded, so it could make a copy of it (using copyseg()). As the process is not now guaranteed to be completely loaded, newproc() must be modified to account for it (by causing each segment to be swapped in and then duplicated). A similar problem exists with shared 'text' segments: the mainstore usage count becomes meaningless.

A method of locking a process so that it will not be scheduled by the CPU must be developed and used any time a process is waiting for a segment to be swapped in. Also the swapping algorithm (sched()) should be modified to process working sets instead of processes. The pager has access to flags which indicate whether a segment has been referenced, or written into. For each process these bits should be sampled periodically and noted. The exact nature of the sampling and use of the results (that is, which paging algorithm should be used) are left to the person doing the implementation, as many tradeoffs exist, and most likely experiments should be performed to determine the best method. For a survey of various paging algorithms see any of the several articles on the subject (for example, Belady (1966) or Denning (1970)).

A final problem area is the required ability to lock pages in memory, either for efficiency (for example, the stack page) or necessity (the profile array must be in mainstore whenever a process with profiling turned on is running). Basically the main problem here is determining when a page should be locked, and providing a data structure to do it. One routine which needs some changes because of the locking problem is the profile primitive (to lock in the required pages).

Most of the major items to be considered in the final stage of converting UNIX to a paging system have been considered. Much latitude has been given to the systems

designer (implementor) in the fourth stage, and yet he is directed to the main problem areas and is given the basic structure of the solutions to the problems. The problems discussed do not cover all of the trivial (or not-so-trivial) details as they would cause the discussion to become too lengthy and pedantic. Also there is little hope of being able to identify all of the potential problem areas, in a task of this magnitude, without actually trying to implement it.

When the four phases have been implemented UNIX will have been converted to a paging (with segments) system. Most of the rewards gained (at least by the latter stages) will be simply in providing a well-structured base for studies into paging systems, which is significant for two major reasons. First UNIX is a very 'clearly' structured operating system, and these phases, if implemented along the lines described in the chapter, will remain 'clean'. This alone would provide a good base for students studying operating system principles. But there is the added feature of it being done on a machine which is affordable, that is, one which is not so large and expensive that hands-on experience must be denied for reasons of economic infeasibility.

Chapter V

Conclusion

5.1 Results

The purpose of the thesis was to examine the feasibility of converting UNIX to a paging system, and also to provide a firm basic understanding of the internal structure of UNIX. As was shown in Chapter IV, several experiments in the area of virtual memory may be performed on UNIX without completely changing the structure of UNIX. For the most part these various experiments, or stages, can be implemented without changing the user's conception or interface to the operating system. Because the user of the system need not be aware of the changes, as various steps are implemented, the volume of software which needs to be altered or rewritten is reduced. Quite often changes in the internal structure of an operating system alter the user interface, requiring large volumes of software to be modified, costing much time and money.

UNIX was originally written in a clean well-structured manner, in a high-level language. The clean structure of UNIX has been maintained in the experiments described in Chapter IV. Because of the inherent structure and modularity of UNIX the aforementioned modifications are easier than they would be on a more conventional loosely-structured system. The advantages of modularity and a clean, well-disciplined operating system (or programs in general) are given in Kernighan (1974) and Dahl (1972) among others.

Perhaps the most important achievement of the thesis is to demonstrate that with some foresight and in-depth analysis of a well-structured operating system, guidelines may be drawn up to modify the system in a manner which is consistent with the original form. The guidelines (or in some cases detailed explanations) which were drawn up in Chapter IV are not to be taken lightly; that is, by following them a programmer could safely modify UNIX in the prescribed manner, but unless they are taken in hand the system, as well as the programmer, would suffer. Much time was spent in comprehending the UNIX system, which is distributed without comments or an internal operations description. Appendix 2 was generated to aid students, programmers and anyone else who is exploring the internal operation of UNIX.

Performing a major modification to a large operating system is not an easy task, no matter how well-written or how modular that system is. Before the modification is actually attempted an in-depth study of the current algorithms and communications structures must be made. Chapter II gave a description of the portions of UNIX which would be affected by the proposed change. For testing purposes and ease of implementation the conversion from a swapping to a paging system was partitioned into four separate stages. After each stage has been implemented a complete and runnable system will have been generated and it should be checked out before continuing on to the remaining stages. The experi-

ments outlined in Chapter IV are not the only possible experiments, as will be shown in the next section.

5.2 Further Research

Although a working knowledge of the whole UNIX system was obtained during the preparation of Appendix 2, only a small fraction of the operating system was explored in detail in the thesis. The reader may have surmised that several other experiments could be performed on the system. The remaining experiments can be broken down into three categories: 1) efficiency, making UNIX more efficient, 2) security, finding and examining security leaks and patching them up, and 3) diversity, extending UNIX to do more for users.

Almost since the concept of an operating system was defined the users of operating systems have clamoured for more efficiency, feeling that their job would run faster on the 'bare' machine. The users were correct (as individuals, not as a group) in almost all cases, including UNIX. Although most of the overhead in an operating system is caused by trying to protect one user from another, usually the overhead is higher than need be. Within UNIX there are several areas which should be investigated, and experiments to correct them generated. When a context switch (i.e. a change of processor mode, and the saving/restoration of the machine state) is done, currently all the registers must be saved and restored. Since the PDP11/45 does not provide an

instruction to save all the registers at once, it must be accomplished slowly by successive store operations. To help alleviate this problem, the designers of the PDP11/45 provided two register sets, switchable by a bit in the PS. Whether UNIX could use the two register sets should be investigated and possibly implemented. The major difficulty is in accessing user parameters to system calls. Another inefficiency in UNIX concerns the fact that it is written in a high-level language. The problem is not the language, but its support by the hardware. Currently, procedure calls are expensive, again because the registers must be saved. Two solutions to the problem could be investigated. First recall that the PDP11 series of machines is capable of handling memories of varying speeds. The register save and restore ('csav()' and 'cret()') could be placed in high-speed store to decrease the time spent executing them. Also other routines which are computationally bound could be placed there. The second solution is more exotic: re-microprogramming the CPU to include the required instructions. This would be an interesting project for the adventurous. Other possible experiments in the area of efficiency include: queuing disk accesses with various algorithms (currently first-come-first-served); overlapping seeks and data transfers (the drives will allow this); modifying the scheduling algorithm; profiling the system to find where it spends its time and examining the code for possible improvements.

Another major aspect of UNIX which could be investigated is that of system security. Although UNIX seems to be relatively secure no proof of the security exists. Because there is no accounting scheme in UNIX (can you smell a project here?) very few users have any reason to 'crash' the system. A contemporary problem in the academic world is proving that certain operating systems are secure; UNIX would be a good candidate for a study of this nature. Several of the array bounds of system structures are not checked when elements are added. (For example coremap[ ] bounds are not checked by mfree() when it is adding free space; if main store becomes sufficiently fragmented the array will overflow its bounds.) Various problems like the array-subscript problem must be discovered and corrected to have a secure, reliable system.

Very little attention was given herein to the diversity of UNIX. When UNIX was written the main criterion was to provide a useable timesharing system. This goal was met with outstanding success. Although UNIX performs very well as a timesharing system there are several difficulties in attempting real-time applications under the system. The difficulties could be defined and the system oriented to solving real-time problems. A few suggestions as to the nature of problems which would be encountered are: a faster, more flexible method of sending interrupts from kernel input/output device drivers to user programs needs to be formulated and implemented; input operations in a user pro-

gram must be conditional in certain cases (i.e. the read() will return with a zero length if no input currently exists); pipes should allow a mechanism for sending end-of-files without closing the pipe; and finally a two-level system structure using the supervisor mode of the machine could be developed to aid real-time applications.

The above suggestions cover only a small number of possible aspects of UNIX which could be explored and improved upon. Since UNIX is an easy-to-use system there will be (and is) an interest in making it more flexible.

References

Alexander, M., (1972), "Organization and features of the Michigan terminal system", AFIPS Conference Proceedings, SJCC, vol. 40, pp. 585-591.

Belady, L., (1966), "A Study of Replacement Algorithms for a Virtual Storage Computer", IBM Systems Journal, vol. 5, no. 2, pp. 282-288.

Dahl, O., Dijkstra, E., and Hoare, C., (1972), Structured Programming, Academic Press, London and New York, 1972.

Denning, P., (1968), "The Working Set Model for Program Behavior", Communications of the ACM, vol. 11, no. 5, pp. 323-333, May.

Denning, P., (1970), "Virtual Memory", Computing Surveys, vol. 2, no. 3, pp. 153-189.

Deutsch, L., and Lampson, B. (1965), "SDS 930 Time-sharing system Preliminary Reference Manual", Doc. 30.10.10, Project GENIE, University of California at Berkeley, April.

Digital Equipment Corporation (1971), Disk Operating System Monitor - Programmers Handbook, DEC-11-OMONA-A-D, Maynard, Massachusetts.

Digital Equipment Corporation (1972), "KT11-C Memory Management Unit Maintenance Manual", DEC-11-HKTB-D, Maynard, Massachusetts.

Digital Equipment Corporation (1975), PDP 11/04/05/10/35/40/45 Processor Handbook, 1975 Edition, Maynard, Massachusetts.

Ferentz, (1975), UNIX Newsletter, vol. 1, no. 1, July, Physics Dept., Brooklyn College of CUNY, Brooklyn, N.Y., 11210.

IBM (1967), IBM System/360 Model 67 Functional Characteristics, Form A27-2719, International Business Machines Corporation, Kingston, New York.

IBM (1973), IBM System/360 Operating System: System Control Blocks, International Business Machines, Form GC28-6628-9, Poughkeepsie, N.Y., 12602.

Kernighan, B., and Plauger, P., (1974), The Elements of Programming Style, Published by McGraw-Hill Inc. for Bell Telephone Laboratories, Incorporated.

Madnick, S., Donovan, J., (1974), Operating Systems, Chapter 3, pp. 105-208, McGraw-Hill, Inc., New York, New York, 1974.

Organick, E, (1973), Computer System Organization, The B5700/B6700 Series, ACM Monograph Series, Academic Press, New York and London, 1973.

Randell, B. and Kuchner, C., (1968), "Dynamic Storage Allocation Systems", CACM, Vol. 11, no. 5, pp. 297-306, May.

Ritchie, D., (1973), "C Reference Manual", published in Documents for Use with the UNIX Time-Sharing System, Sixth Edition, Section 3, pp. 1-30.

Ritchie, D., and Thompson, K., (1974), "The UNIX Time Sharing System", CACM, Vol. 17, no. 7, pp. 365-375, July.

Spier, M. (1973), "A Model Implementation for Protective Domains", International Journal of Computer and Information Sciences, Vol. 2, no. 3, pp. 201-229.

Spier, M., Hastings, T., Cutler, D. (1973a), "An Experimental Implementation of the Kernel/Domain Architecture", Fourth Symposium on Operating System Principles, Operating Systems Review, SIGOPS, Vol. 7, no. 4, pp. 8-21, Oct 1973.

Spooner, C., (1971), "A Software Architecture for the 70's: Part I - The General Approach", Software - Practice & Experience, Vol. 1, no. 1, pp. 5-37.

Thompson, K., and Ritchie, D., (1975), UNIX Programmer's Manual Bell Telephone Laboratories, Sixth Edition, May 1975.

Appendix 1
System Structures

The Proc-structure

One structure is allocated per active process. It contains all data needed about the process while the process may be swapped out. Other per process data (user-structure) is swapped out with the process.

```
struct  proc
{
        char    p_stat;
        char    p_flag;
        char    p_pri;          /* priority, negative is high */
        char    p_sig;          /*signal number sent to process */
        char    p_uid;          /* user id, used to direct tty
                                signals */
        char    p_time;         /* resident time for scheduling */
        int     p_ttyp;         /* controlling tty */
        int     p_pid;          /* unique process id */
        int     p_ppid;         /* process id of parent */
        int     p_addr;         /* address of swappable image */
        int     p_size;         /* size of swappable image */
        int     p_wchan;        /* event process is awaiting */
        int     *p_textp;       /* pointer to text structure */
} proc[NPROC];

/* stat codes */
#define SSLEEP  1               /* sleeping on high priority */
#define SWAIT   2               /* sleeping on low priority */
#define SRUN    3               /* running */
#define SIDL    4               /* intermediate state in
                                process creation */
#define SZOMB   5               /*intermediate state in
                                termination */

/* flag codes */
#define SLOAD   01              /* in core */
#define SSYS    02              /* scheduling process */
#define SLOCK   04              /* process cannot be swapped */
#define SSWAP   010             /*process is being swapped out*/
```

Text-structure

One structure is allocated per pure proceedure (read-only) on the swapping device.

```
struct text
{
        int     x_daddr;        /* disk address of segment */
        int     x_caddr;        /*mainstore address, if loaded */
        int     x_size;         /* size (*64) */
        int     *x_iptr;        /* inode of prototype */
        char    x_count;        /* reference count */
        char    x_ccount;       /* number of loaded references */
} text[NTEXT];
```

Inode-structure

The I node is the focus of all file activity in UNIX. There is a unique inode allocated for each active file, each current directory, each mounted-on file, text file, and the root. An inode is 'named' by its dev/inumber pair.

```
struct  inode
{
        char    i_flag;
        char    i_count;        /* reference count */
        int     i_dev;          /* device where inode resides */
        int     i_number;       /* i number, 1-to-1 with
                                   device address */
        int     i_mode;
        char    i_nlink;        /* directory entries */
        char    i_uid;          /* owner */
        char    i_gid;          /* group of owner */
        char    i_size0;        /* most significant of size */
        char    *i_size1;       /* least sig */
        int     i_addr[8];      /* device addresses consti-
                                   tuting file */
        int     i_lastr;        /* last logical block read
                                   (for read-ahead) */
} inode[NINODE];

/* flags */
#define ILOCK   01              /* inode is locked */
#define IUPD    02              /* inode has been modified */
#define IACC    04              /* inode access time to be
                                   updated */
#define IMOUNT  010             /* inode is mounted on */
#define IWANT   020             /* some process waiting on lock */
#define ITEXT   040             /* inode is pure text prototype */

/* modes */
#define IALLOC  0100000         /* file is used */
#define IFMT    060000          /* type of file */
#define         IFDIR   040000  /* directory */
#define         IFCHR   020000  /* character special */
#define         IFBLK   060000  /* block special, 0 is regular */
#define ILARG   010000          /* large addressing algorithm */
#define ISUID   04000           /* set user id on execution */
#define ISGID   02000           /* set group id on execution */
#define ISVTX   01000           /* save swapped text even after
                                   use */
#define IREAD   0400            /* read, write, execute
                                   permissions */
#define IWRITE  0200
#define IEXEC   0100
```

The User-structure

One user-structure is allocated per process. It contains all of the per process data that doesn't need to be referenced while the process is swapped out. The user-structure is USIZE*64 bytes long; resides at virtual-kernel location 0140000; contains the system stack per process; and points to the proc-structure for the same process.

```
struct user{
        int     u_rsav[2];      /* save r5,r6 when exchanging
                                   stacks */
        int     u_fsav[25];     /* save fp registers */
        char    u_segflg;       /* flag for IO; user or kernel
                                   address space */
        char    u_error;        /* return error code */
        char    u_uid;          /* effective user id */
        char    u_gid;          /* effective group id */
        char    u_ruid;         /* real user id */
        char    u_rgid;         /* real group id */
        int     u_procp;        /* pointer to proc structure */
        char    *u_base;        /* base address for IO */
        char    *u_count;       /* bytes remaining for IO */
        char    *u_offset[2];   /* offset in file for IO */
        int     *u_cdir;        /* pointer to inode of current
                                   directory */
        char    u_dbuf[DIRSIZ]; /* current pathname component */
        char    *u_dirp;        /* current pointer to inode */
        struct  {               /* current directory entry */
                int     u_ino;
                char    u_name[DIRSIZ];
        } u_dent;
        int     *u_pdir;        /* inode of parent directory of
                                   dirp */
        int     u_uisa[8];      /* prototype of segmentation
                                   addresses (PAR's) */
        int     u_uisd[8];      /* prototype of segmentation
                                   descriptors (PDR's) */
        int     u_ofile[NOFILE];/* pointers to file structures
                                   of open files */
        int     u_arg[5];       /* arguments to system call */
        int     u_tsize;        /* text size (*64) */
        int     u_dsize;        /* data size (*64) */
        int     u_ssize;        /* stack size (*64) */
        int     u_qsav[2];      /* label variable for quits and
                                   interrupts */
        int     u_ssav[2];      /* label variable for swapping */
        int     u_signal[NSIG]; /* disposition of signals */
        int     u_utime;        /* this process user time */
        int     u_stime;        /* this process system time */
        int     u_cutime[2];    /* sum of childs' utimes */
```

```
        int     u_cstime[2];    /* sum of childs' stimes */
        int     *u_ar0;         /* address of users saved R0 */
        int     u_prof[4];      /* profile arguments */
        char    u_nice;         /* scheduling parameter */
        char    u_dsleep;       /* another one */
/* kernel stack per user extends from u + USIZE*64
 *      backward not to reach here */
} u;
```

Appendix 2
Kernel Routine Abstracts

The routines which define the kernel of UNIX are described herein. The format of the description is as follows. First the calling sequence is given in standard 'C' notation, with the parameters represented by symbolic names. For example, bmap(*ip,bn) the routine 'bmap' is called with two parameters; '*ip' a pointer to an inode structure, and 'bn'- an integer block-number. Following the calling sequence is a brief description of the function of the particular routine, and its return values. Which file the routine resides in is then provided as an index into the source code, as well as a list of the routines which are called. The UNIX system as distributed has all of the source for the kernel in directory /usr/sys/ken, and they are readable by anyone. The final item in the description is a list of the structures (from /usr/sys) which are referenced (r) or modified (m). The Appendix will be most useful to the student who is attempting to comprehend the UNIX source code for the first time, although it is a helpful cross reference to even the seasoned UNIX guru.

access(*ip,mode)
: If the currently executing user has 'mode' access to file described by inode pointed to by 'ip' (the super user always does) then return(0); otherwise u.u_error <- EACCES, and return(1).
Resides in file 'fio.c'; and calls getfs.
Structures used are user(r,m), filesys(r), and inode(r).

alloc(dev)
: Get a block on device 'dev', return block pointer to memory buffer for block if buffer space exists; otherwise return(0) and set u.u_error <- ENOSPAC.
Resides in file 'alloc.c'; and calls getfs, sleep, bread, bcopy, brelse, badblock, prdev, wakeup, getblk, and clrbuf. Structures used are: filsys(r,m), buf(r,m), and user(m) for error return.

backup()
: Backs up an instruction which has caused a segmentation violation.
Resides in file 'mch.s'; and calls no other routines.

badblock(*fp,bno,dev)
Checks to see if block number 'bno' is within the user limits on device 'dev'. The superblock for 'dev' is pointed to by '*fp'.
Resides in file 'alloc.c'; and calls prdev. The only structure used is filesys(r).

bawrite(*bp)
Starts an output operation described by 'buf' structure pointed to by 'bp'. The completion of the operation is not waited for.
Resides in file 'bio.c'; and calls bwrite. The only structure used is buf(m).

bcopy(*from,*to,count)
Copies 'count' integers (or pointers) from area pointed to by 'from' to area pointed to by 'to'. This should be a machine instruction.
Resides in file 'subr.c'; and calls no other routines.

bdwrite(*bp)
Queues the output operation described by buf structure pointed to by 'bp'. If the operation was for a tape it is started immediatly as tapes are sequential devices.
Resides in file 'bio.c'; and calls bawrite and brelse. The only structure used is buf(r,m).

bflush(dev)
Start writing out all write behind blocks for device 'dev' (or all devices if 'dev' = NODEV).
Resides in file 'bio.c'; and calls spl6, notavail, bwrite and spl0. Buf(r,m) is the only structure used.

binit()
Initialize 'bfreelist' structure, build and initialize buffer pool; clear all open flags for block devices.
Resides in file 'bio.c'; and calls brelse. Structures used are: buf(m), devtab(m), and bdevsw(r).

bmap(*ip,bn)
Places block number 'bn' into inode structure pointed to by 'ip'. This may entail changing inode structure to a large file from a small one. If file grows too large u.u_error <- EFBIG.
Resides in file 'subr.c'; and calls alloc, bdwrite, bread, and brelse. Structures used are: inode(r,m), user(r,m), and buf(r,m).

bread(dev,blkno)
Internal routine to read in block 'blkno' from device 'dev'. A buffer for the data is found and checked to see if it already contains the correct block. If not then 256 words are read in from the device, and the process sleeps until input is complete. A pointer to the buf structure is returned to the caller.
Resides in file 'bio.c'; and calls getblk, *strategy and iowait. The only structure used is buf(r,m).

breada(dev,blkno,rablkno)
Block device read ahead routine. Block 'blkno' is read in from device 'dev' and waited for. The read of block 'rablkno' is started, but not waited for. This is used to speed up sequential inputs, such as directory searches. A pointer to the 'buf' structure is returned to the caller.
Resides in file 'bio.c'; and calls incore, getblk, *strategy, brelse, bread and iowait. The only structure used is buf(r,m).

brelse(*bp)
Rechains buf structure pointed to by 'bp' back on the available list. If processes were waiting for an available buffer they are waked up.
Resides in file 'bio.c'; and calls wakeup and spl6. The only structure used is buf(r,m).

bwrite(*bp)
Writes the buffer pointed to by 'bp' out to the device described by 'bp' buf structure. If not disabled by ASYNC flag, output is waited for and buffer released.
Resides in file 'bio.c'; and calls iowait, *strategy, brelse and geterror. The only structure used is buf(r,m).

chdir()
System primitive to change users current directory to the one specified in u.u_dirp. First desired directory is found and inode inserted in table. Error return (u.u_error <- ENOTDIR) if non existent, not a directory, or not executable by this user. The new inode pointer is entered in u.u_cdir and inode is unlocked.
Resides in file 'sys4.c'; and calls uchar, namei, iput, prele, and access. Structures used are: user(r,m), and inode(r,m).

chmod()

System primitive to change the mode of an inode describing file filename, which is passed as a parameter in u.u_dirp. The mode of a file determines whether one can read, write or execute the file.
Resides in file 'sys4.c'; and calls owner, and iput. Structures used are: inode(m), and user(r).

chown()

System primitive to change the owner of a file. The current user must be the owner of the file in question. The name of the file is pointed to by u.u_dirp, and the new owner is pointed to by u.u_arg[1].
Resides in file 'sys4.c'; and calls owner, suser, and iput. Structures used are: inode(m), user(r).

cinit()

Initialize cfreelist structure, and clear all flags for all of the terminal type devices in the system.
Resides in file 'tty.c'; and calls no other routines. Structures used are: cblock(m), and cdevsw(r)

clearseg(*seg)

Clears the segment (32 words) which is pointed to by 'seg'. 'Seg' is the real segment address (real address/64).
Resides in file 'mch.s'; and calls no other routines.

clock(dev,sp,r1,nps,r0,pc,ps)

Routine is entered via a clock interrupt. Functions: keeps track of clock times; updates front panel light display; does callouts (executes certain functions after timer queue elements expire); updates profile; updates user time, system time, and real time; wakesup 'lbolts'; wakesup 'runin' if sleeping; if in user mode switchs tasks; increments priority to max of 120.
Resides in file 'clock.c'; and calls display, incupc, spl1, spl5, wakeup, issig, psig, savfp, and swtch. Structures used are: callout(r,m), lbolt(m), time(m), callo(r,m), user(r,m), proc(r,m), and runin.

close()

System entry for close[II] primitive. Closes file pointed to by file descriptor index passed in u.u_ar0[r0]. Major work is done by closef(fio.c).
Resides in file 'sys2.c'; and calls getf, and

closef. Structures used are: user(r,m), and file(r).

closef(*fp)
Close file described by 'file' structure pointed to by 'fp'. Closing decrements count of closes and removes inode if count indicates no more users of this file. If file was a pipe, changes mode of inode to indicate not read or write, and wakes up i_cnt and i_dev.
Resides in file 'fio.c'; and calls closei, and wakeup. Structures used are: file(r,m), and inode(m).

closei(*ip,rw)
Calls the close routine of device pointed to by inode structure 'ip'. 'rw' is the second argument passed to the device's close routine. The inode is then removed from the inode table by iput.
Resides in file 'fio.c'; and calls iput. Structures used are: cdevsw(r), inode(r), and bdevsw(r).

clrbuf(*bp)
Buffer pointed to by buf structure 'bp' is cleared to zero.
Resides in file 'bio.c'; and calls no other routines. The only structure used is buf(r).

copyin(*a1,*a2,len)
Copies 'len' bytes from area pointed to by 'a1' in user space, to area pointed to by 'a2' in kernel space.
Resides in file 'mch.s'; and calls no other routines.

copyout(*a1,*a2,len)
Copies 'len' bytes from area pointed to by 'a1' in kernel space, to area pointed to by 'a2' in user space.
Resides in file 'mch.s'; and calls no other routines.

copyseg(*a1,*a2)
Copies a segment (32 words) from area pointed to by 'a1' to area pointed to by 'a2'. Both addresses are considered as real segment addresses (real address/64). On the 11/45 this move is performed through the 'system' segmentation registers.
Resides in file 'mch.s'; and calls no other routines.

core()

Makes a core image on file 'core' of current directory if possible and returns(1), else return zero. The order of a core image is user structure followed by the core image excluding the non writable text portion.
Resides in file 'sig.c'; and calls estabur, writei, itrunc, access, maknode, namei, and iput. Structures used are: user(r,m), proc(r).

cpass()

Returns a character from the current user area, pointed to by u.u_base, or returns -1 if u.u_count > 0 or if a memory fault occurred during character fetch. If faulted then u.u_error <- EFAULT.
Resides in file 'subr.c'; and calls fubyte. The only structure used is user(r,m).

creat()

System entry for creation of a file. Creates a file of name pointed to by u.u_dirp of mode u.u_arg[1]. File is also opened for writing and truncated. File doesn't have to be writable to be written on unless it existed before call.
Resides in file 'sys2.c'; and calls namei, uchar, maknode, and open1. The only structure used is user(r).

deverror(*bp,oct)

Prints on IPL console the device error information from the 'buf' structure pointed to by 'bp', and device error 'oct'.
Resides in file 'prf.c'; and calls prdev,and printf. The only structure used is buf(r).

devstart(*bp,devloc,devblk,hbcom)

An I/O operation is started on the disk at address 'devloc'. The operation is described by buf structure 'bp', and takes place on block 'devblk'.
Resides in file 'bio.c'; and calls no other routines. The only structure used is buf(r).

display()

Displays on the front panel lights the contents of word addressed by the console switches. If bit zero is off then display kernel mode word, else display user addressed word.
Resides in file 'mch.s'; and calls no other routines.

dpadd(*dw,sw)
Adds the single word 'sw' to the double word pointed to by 'dw'.
Resides in file 'mch.s'; and calls no other routines.

dpcmp(a1,a2,b1,b2)
The double words (a1,a2) and (b1,b2) are compared. Returns +512, -512, or 0 depending on the result of the comparison.
Resides in file 'mch.s'; and calls no other routines.

dup()
System primitive to duplicate open file descriptors. Basically this routine simply locates the first empty slot in u.u_ofile[ ] and stores the address of the file structure pointed to by u.u_arg[1] in it.
Resides in file 'sys3.c'; and calls getf and ufalloc. Structures used are: file(m) and user(r,m).

estabur(nt,nd,ns)
The arguments are the size of the text, data, and stack respectively. A check is made to see if the process described by nt, nd, and ns will fit in the available addressing space, then offsets are generated for the segment table. The pieces are fitted together in the following order: text(negative offset from p_addr), u.segment, data segment, and the stack is placed at the top of the address space, expanding downward. If the process will not fit then u.u_error <- ENOMEM.
Resides in file 'main.c'; and calls nseg, and sureg. Segments used are: user(r,m).

exec()
An implementation of the exec(II) system primitive, as described in the UNIX manual. Removes old memory image, gets new one, and starts it running at location zero.
Resides in file 'sys1.c'; and calls namei, sleep, getblk, access, fuword, fubyte, readi, estabur, xfree, xalloc, expand, clearseg, suword, subyte, iput, wakeup, and brelse. Structures used are: inode(r,m), user(r,m), execnt(r,m), and buf(r).

exit()
Exit from a process with no return. Closes all open files, clears signal entries, frees all memory except 'user' structure and wakes up any processes waiting for the completion of this process. When the waiting process is wakened this

process is removed from the process table (see wait).
Resides in file 'sys1.c'; and calls closef, iput, xfree, swtch, malloc, getblk, bcopy, bwrite, mfree, wakeup, and panic (if there is no init process). Structures used are user(r,m), and proc(r,m).

expand(newsize)

Expands the current process to size 'newsize'. This is done with an in memory copy if there is enough room, otherwise it swaps the image to disk with new size. If the new size is smaller than the old size then the difference is freed.
Resides in file 'slp.c'; and calls mfree, savu, malloc, xswap, swtch, copyseg, retu and sureg. Structures used are proc(r,m), and user(r,m).

falloc()

Get space in file descriptor array maintained by the system. If an entry is found then initialize it and return pointer to it. Otherwise table is full so report to main console, return null, and u.u_error <- ENFILE.
Resides in file 'fio.c'; and calls ufalloc, and printf. Structures used are user(m) and file(r,m).

fork()

System entry to create a new process (if room, else error (u.u_error <- EAGAIN) return) and returns from both new process and the old process. Returns child's process id to parent and child. Parent returns to 'pc'+2, child returns to 'pc'.
Resides in file 'sys1.c'; and calls newproc. Structures used are: proc(r), user(m).

free(dev,blkaddr)

Adds block 'blkaddr' to free list on device 'dev', set flag for future update of super-block on that device.
Resides in file 'alloc.c'; and calls badblock, getblk, getfs, sleep, bcopy, bwrite, and wakeup. Structures used are: filsys(r,m), buf(r,m), and inode(r,m).

fstat()

System primitive to return, in user provided buffer pointed to by u.u_arg[0], the inode structure of file 'file descriptor' passed in r0.
Resides in file 'sys3.c'; and calls getf, and stat1. Structures used are: user(r), and file(r).

fubyte(v_address)
Pick up the byte at user address 'v_address'; if this address exists then return the word found; else return -1.
Resides in file 'mch.s'; and calls no other routines.

fuword(v_address)
Pick up a word at user address 'v_address'; if this address exists return the word found; else return -1.
Resides in file 'mch.s'; and calls no other routines.

getblk(dev,blkno)
A buffer is obtained to do I/O on device 'dev'. A pointer to the describing buf structure is returned. Buffers for device 'dev' are checked to see if block 'blkno' already resides in main store. If this is the case, it is returned, else, a buffer is obtained from the freelist. This buffer is rechained to the device and its pointer returned.
Resides in file 'bio.c'; and calls spl6, sleep, spl0, notavail and bwrite. The only structure used is buf(r,m).

getc(*qe)
Returns the character (eight bits) from the next queue element pointed to by 'qe'. Returns -1 if the queue is empty.
Resides in file 'mch.s'; and calls no other routines. Uses structure clist(m).

geterror(*bp)
The 'user' error indication is turned on if an error has occurred during the I/O operation described by buf structure pointed to by 'bp'.
Resides in file 'bio.c'; and calls no other routines. Structures used are buf(r) and user(m).

getf(f)
Return address of file descriptor for open file number 'f'. Return zero, and set u.u_error <- EBADF if this file number is not open.
Resides in file 'fio.c'; and calls no other routines. The only structure used by this routine is user(r,m).

getfs(dev)

Get file system descriptor from mount table for device 'dev'.
Resides in file 'alloc.c'; and calls prdev and panic. Structures used are: mount(r,m), filsys(r,m) and buf(r).

getgid()

System primitive which returns current and real group identification number from u.u_gid and u.u_rgid, to the user in r0.
Resides in file 'sys3.c'; and calls no other routines. The only structure used is user(r,m).

getmdev()

Internal subroutine used by smount(sys3.c) and sumount(sys3.c) to obtain device number of the device to be mounted or dismounted.
Resides in file 'sys3.c'; and calls uchar, namei, and iput. Structures used are: inode(r), and user(r,m).

getpid()

System primitive which returns the process id of the calling process.
Resides in file 'sys4.c'; and calls no other routines. The only structure used is user(r,m).

getswit()

System primitive which returns the contents of the switch register to the user in r0.
Resides in file 'sys3.c'; and calls no other routines. The only structure used is user(m).

getuid()

System primitive which returns current and real user identification number from u.u_uid and u.u_ruid, to the user in r0.
Resides in file 'sys3.c'; and calls no other routines. The only structure used is user(r,m).

gtime()

System primitive which returns the time of day in r0, and r1.
Resides in file 'sys4.c'; and calls no other routines. Structures used are: user(m) and time(r).

ialloc(dev)

Get an inode structure (a free inode) from device 'dev'. When no more free inodes exist on that device a panic will be initiated.
Resides in file 'alloc.c'; and calls getfs, sleep, iget, printf, iput, bread, brelse, panic, and wakeup. Structures used are: filsys(r,m), inode(r,m), and buf(r).

idle()

Wait loop for machine when nothing else is being done. If the wait instruction completes, idle returns to caller.
Resides in file 'mch.s'; and calls no other routines.

ifree(dev,ino)

Remove inode 'ino' from device 'dev'; add this inode to list of free inodes on device if list not full, and device is not busy.
Resides in file 'alloc.c'; and calls getfs. The only structure used is : filsys(m).

iget(maj.dev,ino)

Returns pointer to inode structure of inode 'ino' on device 'maj.dev'. If inode sought is not in inode table then space for it is acquired, and it is read in from disk. Variable Maxip is in routine for monitoring purposes only.
Resides in file 'iget.c'; and calls sleep, panic, bread, lrem, and brelse. Structures used are: inode(r,m), mount(r) and buf(r,m).

iinit()

Mount root device, clean up super- block, and get permanent buffer for super- block in user memory.
Resides in file 'alloc.c'; and calls bread, getblk, bcopy, panic, and brelse. Structures used are: user(r), mount(m), time(m), and filsys(r,m).

incore(dev,blkno)

Checks buffer queues for device 'dev' to see if block number 'blkno' is already in memory. If it is then its buffer pointer is returned.
Resides in file 'bio.c'; and calls no other routines. Structures used are bdevsw(r) and buf(r).

incupc(pc,*prof)

Increments the member of the profile array which corresponds to 'pc', if the element is within range, as defined by 'prof'.
Resides in file 'mch.s'; and calls no other

routines. The only structure used is user(r).

iodone(*bp)

Signals that an I/O operation has been completed by setting the done flag in buf structure pointed to by 'bp'. If the initiating process is waiting for I/O it is waked up.
Resides in file 'bio.c'; and calls brelse and wakeup. The only structure used is buf(r,m).

iomove(*bp,off,an,flag)

Copies information to and from supervisor buffers for readi, and writei routines. 'off' is offset from the start of buffer pointed to by 'bp', 'an' number of bytes are transferred. 'flag' determines direction of transfer.
Resides in file 'rdwri.c'; and calls copyin, copyout, dpadd, cpass, and passc. Structures used are: buf(r), and user(r,m).

iowait(*bp)

Waits for the previously initiated I/O described by buf structure pointed to by 'bp'.
Resides in file 'bio.c'; and calls spl6, sleep, spl0 and geterror. The only structure used is buf(r).

iput(*ip)

Releases an inode, pointed to by 'ip', from the inode table. This should be done after the use of the node has ceased so the inode table will not get full. Inode is only removed if 'i_count' decrements to zero. File is removed from device by trunc(), if the number of links to it are zero.
Resides in file 'iget.c'; and calls itrunc, ifree, iupdat, and prele. The only structure referenced is inode(r,m).

issig()

Checks if the current signal in p_sig is enabled. Returns signal number if enabled, zero if not.
Resides in file 'sig.c'; and calls no other routine. Structures used are: proc(r), user(r).

itrunc(*ip)

Remove the contents of a file from disk. Directory entry and inode describing it are left on the disk. Set file to zero length.
Resides in file 'iget.c'; and calls bread, free, and brelse. Structures used are: inode(r,m) and buf(r,m).

iupdat(*ip,*time)
Copies inode pointed to by 'ip' out to the device it resides on with updated access/update times.
Resides in file 'iget.c'; and calls getfs, bread, ldiv, lrem, and bwrite. Structures used are: inode(r,m), and buf(r,m).

kill()
System primitive to send a signal to a process. The UID's of the sending and receiving processes must be the same or the sending process must be the super user, for the primitive to have any effect. The signal to be sent is passed in u.u_arg[0]
Resides in file 'sys4.c'; and calls suser, and psignal. Structures used are: proc(r), and user(r,m).

ldiv(hw,lw,div)
Returns the quotient of the 32 bit concatenation of 'hw.lw' divided by 'div'.
Resides in file 'mch.s'; and calls no other routines.

link()
System entry to make a new link to an existing file. Routine looks up inode of file designated in u.u_dirp, and verifies o.k. If the link name is found or other errors happen u.u_error <- EEXIST. If no errors are found then increment link count of existing file and add entry to specified directory.
Resides in file 'sys2.c'; and calls uchar, namei, suser, iput, and wdir. Structures used are: user(r,m), and inode(r,m).

lrem(hw,lw,div)
Returns the remainder of the 32 bit concatenation of 'hw.lw' divided by 'div'.
Resides in file 'mch.s'; and calls no other routines.

main()
First routine entered at IPL time. Initializes the system, clears memory, reports size of memory, initializes swap space, starts clock, sets up first process, sets up init process and executes it.
Resides in file 'main.c'; and calls clearseg, printf, mfree, fuword, sureg, cinit, binit, iinit, iget, newproc, copyout, sched, and expand. Structures used are: user(m), proc(m), rootdir(m), and inode(m).

maknode(mode)

Makes a node in the directory system. Builds an inode of mode 'mode' and enters name/inode in directory u.u_pdir.
Resides in file 'iget.c'; and calls ialloc, and wdir. Structures used are: user(r), and inode(r,m).

malloc(*map,size)

Finds in map pointed to by 'map' a block of storage of size 'size', deletes it from the free list in map, and returns its address (real address/64). If no space can be found then zero is returned.
Resides in file 'malloc.c'; and calls no other routines. The only structure used is map(r,m).

max(*a,*b)

Returns the maximum of the characters pointed to by 'a' and 'b'.
Resides in file 'rdwri.c'; and calls no other routines.

mfree(*map,size,addr)

Keeps a list (map which is pointed to by 'map') of free storage. This subroutine adds size 'size' blocks to location address 'addr' (given in real address/64). These are kept in ascending order and adjacent blocks are merged.
Resides in file 'malloc.c'; and calls no other routines. The only structure used is map(r,m).

min(*a,*b)

Returns the minimum of the characters pointed to by 'a' and 'b'.
Resides in file 'rdwri.c'; and calls no other routines.

mknod()

System entry for mknod[II] primitive. Checks to see that the caller is super user, and that the file does not already exist. If an error is found u.u_error <- EEXIST. If everything is all right then maknode is called to make the node. The name of the new node is passed in u.u_dirp.
Resides in file 'sys2.c'; and calls uchar, suser, namei, maknode, and iput. Structures used are: user(r,m) and inode(r,m).

namei(*func,flag)

An internal system routine to get an inode for file name retrieved by executing function 'func'. This function is usually either 'schar' (for system names) or 'uchar' (for names in user space). 'flag'=0 if name is sought; 1 if name is to be created; and 2 if name is to be deleted. Return is incremented, locked inode or null if name is not found.
Resides in file 'namei.c'; and calls iget, access, ldiv, brelse, bread, bmap, bcopy, and iput. Structures used are: inode(r,m), user(r,m), and buf(r,m).

newproc()

Generates a new process as a copy of the current process. The new copy is placed either in memory if room or is swapped onto the disk. Returns twice, once for parent process (return a zero) and once for the child (return a one).
Resides in file 'slp.c'; and calls panic, savu, malloc, xswap, and copyseg. Structures used are: user(r,m), text(r,m), proc(r,m), file(m).

nice()

System primitive for setting process priority. Users (unless super user) may only decrease priority by giving a positive number in r0.
Resides in file 'sys4.c'; and calls suser. The only structure used is user(r,m).

nodev()

Sets u.u_error <- NODEV and returns. Used for entries in conf.c (cdevsw and bdevsw) if this type of access is always an error.
Resides in file 'subr.c'; and calls no other routines. The only structure used is user(m).

nosys()

Called if a user attempts an illegal system call (one that does not exist). Prints out primitive number of attempt on ipl console, and returns with u.u_error <- 100.
Resides in file 'trap.c'; and calls printf. The only structure used by this routine is user(m).

notavail(*bp)

Unchains the buffer pointed to by 'bp' from the available list.
Resides in file 'bio.c'; and calls spl6. The only structure used is buf(r,m).

nseg(n)

Returns the number of segments in 'n' bytes; (n+127)>>7.
Resides in file 'main.c'; and calls no other routines.

nulldev()

Does nothing but return. Used for null entries in conf.c (bdevsw and cdevsw).
Resides in file 'subr.c'; and calls no other routines.

nullsys()

Called if system routine is a no operation, this routine does nothing. This is usually used for old system entry points which no longer exist.
Resides in file 'trap.c'; and calls no other routines.

open()

System entry for opening an inode. This is the user interface and reads in inode describing file of name pointed to by u.u_dirp. This inode is then passed to open1 with a mode of u.u_arg[1]+1.
Resides in file 'sys2.c'; and calls namei, open1, and uchar. The only structure used is user(r,m).

open1(*ip,mode,trf)

Opens (via openi) inode pointed to by 'ip' . This inode is opened with mode 'mode'. If 'trf' is 2 (i.e. just created) then check mode to see if o.k., initialize inode and fill file structure. If 'trf' is zero then truncate length of file to zero before initializing inode and filling file structure.
Resides in file 'sys2.c'; and calls access, openi, iput, itrunc, prele, and falloc. Structures used are: inode(r,m), file(m), and user(r).

openi(*ip,rw)

Calls open routine for device pointed to by inode structure pointed to by 'ip'. 'rw' is passed as second argument to open routine ('dev' is the first argument).
Resides in file 'fio.c'; and calls device open routine via c(b)devsw. Structures used are: inode(r), user(m), cdevsw(r), and bdevsw(r).

owner()

Checks to see if this user owns file named in u.u_dirp. Return one if user is owner or super user, zero in all other cases.
Resides in file 'fio.c'; and calls suser, namei

and iput. Structures used are user(r) and inode(r).

panic(*s)

Outputs panic message to ipl console and appends string pointed to by 's'. The update function is then performed, and the kernel stack pointer is saved in variable 'kisa6' and finally the machine is placed in an endless idle loop.
Resides in file 'prf.c'; and calls update, printf, and idle.

passc(c)

Places the character 'c' in the current user area, updating u.u_base, u.u_count, and u.u_offset[ ]. Returns zero if everything is all right, and returns -1 if count is exausted or a memory fault occurs, in the latter case u.u_error <- EFAULT.
Resides in file 'subr.c'; and calls subyte. The only structure used is user(r,m).

physio(*strat,*bp,dev,rw)

A nonstandard length (°256 words) I/O operation is performed on device 'dev'. The routine actually used to do the I/O is pointed to by 'strat'. The parameters used in the operation are passed to this routine in the user structure.
Resides in file 'bio.c'; and calls spl6, sleep, lshift, *strat, wakeup and geterror. Structures used are user(r) and buf(m).

pipe()

System primitive to set up input/output pipe descriptors for a process. An 'inode' is allocated on 'rootdev', and file descriptors are generated for input and output. The user file table for these descriptors is returned to the user in registers r1(write) and r0(read).
Resides in file 'pipe.c'; and calls ialloc, falloc and iput. Structures used are: inode(m), user(m) and file(m).

plock(*ip)

Internal routine to lock out multiple use of a common pipe pointed to by inode 'ip'. The calling process will wait until 'inode' is unlocked; then it will lock it to keep out others.
Resides in file 'pipe.c'; and calls sleep. The only structure used is inode(r,m).

prdev(*str,dev)
Prints string pointed to by 'str' along with major/minor device 'dev' on IPL console.
Resides in file 'prf.c'; and calls printf. Uses structure dev(r).

prele(*ip)
Pipe unlocking routine. Undoes a 'plock' on inode pointed to by 'ip'.
Resides in file 'pipe.c'; and calls wakeup. The only structure used is: inode(r,m).

printf(*fmt,x1,x2,...,x9,xa,xb,xc)
This is a limited version of the more general printf routine used by users. This version includes format codes d,l,o,c, and s. There is a maximum of twelve arguments. It is used to print urgent messages on the IPL terminal.
Resides in file 'prf.c'; and calls putchar, and printn.

printn(n,b)
Print number 'n' in base 'b', strip leading zeroes.
Resides in file 'prf.c'; and calls printn, ldiv, and putchar.

profil()
System primitive to set up and turn on profiling of the program counter. U.u_prof[ ] are initialized from user arguments u.u_arg[ ].
Resides in file 'sys4.c'; and calls no other routines. The only structure used is user(r,m).

psig()
Issue signal p_sig to the current process. If the user has trapped signal then save current pc and ps and return to user at subroutine address specified in the trap. If not trapped dump memory if appropriate. If core is dumped then add 200 to signal number and return (exit to another process).
Resides in file 'sig.c'; and calls suword, core, and exit. Structures used are: user(r,m), and proc(m).

psignal(*p,sig)
Issues signal 'sig' to process pointed to by 'p'. If process is waiting (p_stat == SWAIT) then set p_stat to SRUN, clear p_wchan, and wakeup all process' waiting on 'runout'.
Resides in file 'sig.c'; and calls wakeup. Structures used are: proc(r,m), and runout(r,m).

putc(c,*qe)

Places character 'c' onto queue described by 'qe'. Returns zero if o.k., non zero if there was a problem, such as list full.
Resides in file 'mch.s'; and calls no other routines. The only structure used is clist(m).

putchar(c)

Outputs character 'c' to ipl console terminal, unless switch register is all zeroes.
Resides in file 'prf.c'; and calls putchar.

rdwr(mode)

Places I/O parameters fetched from file descriptor structure, into user structure. Either the inode read write handler (readi, writei, depending on 'mode'), or the pipe handler (readp, writep, depending on 'mode') is called.
Resides in file 'sys2.c'; and calls getf, readp, writep, readi, writei, and dpadd. Structures used are file(r,m), and user(r,m).

read()

System entry for reading from I/O units. All this routine does is call rdwr with a mode of read.
Resides in file 'sys2.c'; and calls rdwr.

readi(*ip)

Reads from device (file) described by inode pointed to by 'ip'. If the device is a character type then one line is returned by calling indirectly (through cdevsw) the device handler. If the device is a block device enough blocks are read to satisfy the count in u.u_count, or until an end-of-file is reached; this is done by 'bread'.
Resides in file 'rdwri.c'; and calls breada, lshift, min, dpcmp, bmap, bread, iomove, and brelse. Structures used are: user(r,m), inode(r), buf(r), and cdevsw(r).

readp(*fp)

Internal routine to read a pipe described by 'file' structure pointed to by 'fp'. If the pipe is empty the process will wait until data is entered (writep). The available data will then be returned to the caller until his buffer is full, or the pipe becomes empty.
Resides in file 'pipe.c'; and calls plock, wakeup, sleep, readi and prele. Structures used are: file(r,m), inode(r,m) and user(r,m).

retu(*u)

Changes current user pointer (Virtual address 0140000) to be the pointer to 'u'. Switches the current user of the cpu.
Resides in file 'mch.s'; and calls no other routines. The only structure used is user(r).

rexit()

System primitive entry point of 'exit'. Calls 'exit', to terminate the current process, after fetching the return code from user register zero.
Resides in file 'sys1.c'; and calls exit. The only structure used is user(r).

savfp()

Saves the floating point registers of the current user, in u.u_fsav[25]. This is only done when there is to be a process switch after an interrupt.
Resides in file 'mch.s'; and calls no other routines. The only structure used is user(m).

savu(*u)

Saves stack pointer, and r5 (local stack pointer) into 'user' structure pointed to by 'u'.
Resides in file 'mch.s'; and calls no other routines. The only structure used is user(m).

sbreak()

System primitive (break[II]) to set the top location of the current process' data area to the value passed in u.u_arg[0]. This routine allocates, and deallocates memory as needed. To do this the stack must be copied to a new physical position in memory, as it is always adjacent, physically, to the data area.
Resides in file 'sys1.c'; and calls nseg, estabur, copyseg, expand, and clearseg. Structures used are: user(r,m), and proc(r,m).

schar()

Returns the next character of directory name pointed to by u.u_dirp. This character is fetched from kernel space.
Resides in file 'namei.c'; and calls no other routines. The only structure used is user(r,m).

sched()

Schedules a new process to be run on the cpu, swaps process' into and out of memory if required. Algorithm used is longest in -> out; longest out -> in; no priorities are examined at swap time. Usually called in conjunction with wakeup of 'ru-

nin'. This routine is concerned only with keeping memory full of process' ready to run. It doesn't switch between running process'.
Resides in file 'slp.c'; and calls sleep, malloc, spl6, spl0, xswap, swap, mfree, and panic. Structures used are: proc(r,m), runin(r,m), and text(r,m).

seek()

System entry for seek[II] primitive. Seek adjusts offsets, and sizes of inode and file structures causing the next I/O to reposition the device. If device is pipe then u.u_error <- ESPIPE.
Resides in file 'sys2.c'; and calls getf, inode(r,m) and dpadd. Structures used are: file(r,m), and user(r,m).

setgid()

System primitive which sets the current process' group id, and real group id (in u.u_gid, u.u_rgid) to the contents of r0. If the user is not the super user, and if the argument is not the real group id, then no setting occurs.
Resides in file 'sys3.c'; and calls suser. The only structure used is user(r,m).

setuid()

System primitive which sets the current process' user id, and real user id (in u.u_uid, u.u_uid) to the contents of r0. If the user is not the super user, and if the argument is not the real user id, then no setting occurs.
Resides in file 'sys3.c'; and calls suser. The only structure used is user(r,m).

signal(ntp,sig)

Issues signal 'sig' to all process' with teletype number 'ntp'.
Resides in file 'sig.c'; and calls psignal. The only structure used is proc(r,m).

sleep(*chan,prio)

Places current process in the dormant state until a 'wakeup' is performed on variable pointed to by 'chan', at which time the process will be rescheduled at priority 'prio'. As sleep stores information in process structures a single process should never be put to sleep more than once, which is possible in interrupt handlers.
Resides in file 'slp.c'; and calls issig, spl0, wakeup, aretu and swtch. Structures used are: runin(r,m) user(m), and proc(m).

smdate()

System primitive to set the last-modified-date field for file u.u_dirp. The date used is passed in r0, r1 from the caller.
Resides in file 'sys4.c'; and calls owner, iupdat, and iput. Structures used are: user(r) and inode(m).

smount()

System primitive to mount a special file (device) on a file named in u.u_dirp. Device cannot be already mounted, nor can the file on which it is to be mounted be a special file, or in use at this time. After an entry in the mount table is found and initialized, the super block is read in and initialized. Finally the inode is marked as mounted and not locked.
Resides in file 'sys3.c'; and calls uchar, getmdev, namei, bread, brelse, bcopy, iput, prele and getblk. Structures used are: user(r,m), filsys(r,m), mount(r,m), buf(r) and inode(r,m).

spl?()

Set system (machine) priority level to '?'. Priority zero allows all interrupts, and priority seven does not allow any.
Resides in file 'mch.s'; and calls no other routines.

ssig()

System primitive (signal[II]) to set up for catching or ignoring signals. This routine replaces u.u_signal[u.u_arg[0]] with u.u_arg[1], if signal is valid and not Kill signal. The old value of u.u_signal[ ] is returned to the user.
Resides in file 'sys4.c'; and calls no other routines. Structures used are: user(r,m), and proc(m).

sslep()

System primitive (sleep[II]) forcing the current process into a dormant state for r0 seconds. This user is queued up on tout[ ] and then sleep[slp.c] is called with tout as wait event.
Resides in file 'sys2.c'; and calls spl7, spl0, dpcmp, sleep, and dpadd. Structures used are: time(r), user(r), and tout(r,m).

stat()

System primitive (stat[II]) to return, in user provided buffer pointed to by u.u_arg[1], the contents of the inode describing the file pointed to by u.u_arg[0].

Resides in file 'sys3.c'; and calls uchar, namei, stat1, and iput. Structures used are: user(r), and inode(r).

stat1()

This internal routine does the actual information copy for routines fstat[sys3.c] and stat[sys3.c]. Resides in file 'sys3.c'; and calls iupdat, bread, lrem, ldiv, suword, and brelse. Structures used are: buf(r,m), time(r) and inode(r).

stime()

System primitive (stime[II]) which sets the time of day clock to the value passed in r0-r1, if and only if the user is the super user.
Resides in file 'sys3.c'; and calls suser, and wakeup. Structures used are: time(m), and user(r).

subyte(*va,byte)

Store byte 'byte' at address 'va', in user space. If location exists return zero, else -1.
Resides in file 'mch.s'; and calls no other routines.

sumount()

System primitive (umount[II]) to unmount a special file pointed to by u.u_arg[0]. After checking to make sure device was mounted and not busy, the inode pointing to the file which the device was mounted on is marked as not mounted, and it is released. The buffer for the super block is also released.
Resides in file 'sys3.c'; and calls updat, getmdev, iput, and brelse. Structures used are: user(r,m), inode(r,m), and mount(r,m).

sureg()

Sets up user's segment registers from u.u_uisa[8] + p_addr, and also user segment descriptor from u.u_uisd[8].
Resides in file 'main.c'; and calls no other routines. Structures used are: user(r,m), proc(r,m), and text(r).

suser()

Returns one if current user is super user, else return zero and u.u_error <- EPERM.
Resides in file 'fio.c'; and calls no other routines. The only structure used is user(r,w).

suword(*va,word)
Internal routine to store word 'word' in address 'va' in user space. If the address exists then return zero, else return -1.
Resides in file 'mch.s'; and calls no other routines.

swap(blkno,*coreaddr,count,rdflg)
Performs the I/O required to swap a process in or out of memory. The direction is determined by 'rdflg' and takes place with 'count' words being transferred between block 'blkno' and memory address 'coreaddr'. Returns non zero if an I/O error occurred.
Resides in file 'bio.c'; and calls sleep, *strategy and wakeup. The only structure used is buf(m).

swtch()
Do a process switch to the highest priority loaded process wishing to run. Round robin scheduling is performed on process' of the same priority.
Resides in file 'slp.c'; and calls savu, retu, idle, aretu and sureg. Structures used are: proc(r,m), and user(r,m).

sync()
System primitive (sync[II]) which causes an update of super block on all mounted devices.
Resides in file 'sys4.c'; and calls update.

sysent[ ]
A table of the addresses of the system entry points and their argument counts. This table is used by 'trap' to call system primitives.
Resides in file 'sysent.c'; and calls everything.

timeout(*fun,arg,time)
To place an item on the callout list. When time 'time' has expired function 'fun' will be called with argument 'arg'. BEWARE: callout array is <u>not</u> checked for overflow!!!!
Resides in file 'clock.c'; and calls display, incupc, wakeup, psig, issig, savfp, and swtch. Structures used are callo(r,m) and callout(r,m).

times()
System primitive (times[II]) which returns, in user buffer pointed to by u.u_arg[0], real, user, and system times used by the calling process.
Resides in file 'sys4.c'; and calls suword. The only structure used is user(r,m).

trap(dev,sp,r1,nps,r0,pc,ps)

Handles all processor traps. 'Dev' is the trap number (0 =bus error, ..., 9= segment violation); 'sp' is old stack pointer; 'r1' is old r1; the new ps is in 'nps'; 'r0' is the old r0; 'pc' and 'ps' are the old pc and ps respectively. This routine does error checking, signal sending, and parameter gathering for the various types of traps. It also forces a process switch if 15 system calls have been made by the current process without an intervening sleep.
Resides in file 'trap.c'; and calls savfp, psignal, fuword, backup, estabur, expand, copyseg, sysent, idle, clearseg, savu, issig, psig, swtch, printf, panic, and trap1. Structures used are: proc(r,m), and user(r,m).

trap1(*f())

Calls system primitive passed as an argument. This call may be incorporated into trap.c at a future date.
Resides in file 'trap.c'; and calls savu.

uchar()

Returns next character of directory name pointed to by u.u_dirp. This character is fetched from user space, and is returned as -1 if there was a memory fault trap taken because of the fetch.
Resides in file 'namei.c'; and calls fubyte. The only structure used is user(r,m).

ufalloc()

Find an empty slot in user's open file array, u.u_ofile. If none can be found (max of 15) then returns -1 and sets u.u_error <- EMFILE; else returns slot number.
Resides in file 'fio.c'; and calls no other routines. The only structure used is user(r,m).

unlink()

System primitive (unlink[II]) to unlink a directory entry from an inode. Inode number of directory entry is cleared to zero, indicating free directory entry. Link count of inode is decremented, and if count goes to zero the inode and its associated blocks are freed. The name of the file is passed in u.u_dirp, and it will only be unlinked if the current user has the proper permission.
Resides in file 'sys4.c'; and calls prele, uchar, namei, iget, panic, suser, writei, and iput. Structures used are: user(r,m), and inode(r,m).

update()

Write out super block for each mounted device. Write out each modified inode in the inode table. Resides in file 'alloc.c'; and calls bwrite, iupdat, prele, bflush, getblk, and bcopy. Structures used are: updlock(r,m), buf(r), mount(r), inode(r,m), and filsys(r,m).

wait()

System primitive (wait[II]) to wait for the completion of one of the children of a process. If no children exist then return u.u_error <- ECHILD. Removes process table entry for child when it dies, and updates process times of parent to indicate time used by the child. Also frees up 'user' structure left on disk by 'exit()'.
Resides in file 'sys1.c'; and calls bread, mfree, dpadd, brelse, and sleep. Structures used are: proc(r,m), buf(r), and user(r,m).

wakeup(*chan)

Wakes up all process' sleeping on variable 'chan'. This is done through a linear search of the proc[ ] table. When a process is awakened, 'runout' is also awakened, causing the scheduler to schedule this process if it is doing nothing else.
Resides in file 'slp.c'; and calls wakeup. Structures used are: proc(r,m), runrun(m), and runout(r,m).

wdir(*ip)

Makes a directory entry for inode pointed to by 'ip', and releases inode 'ip' from memory buffer.
Resides in file 'iget.c'; and calls writei, and iput. Structures used are: user(r,m), and inode(r).

write()

System primitive (write[II]) to write to I/O devices or pipes. Calls rdwr with a mode of write to do actual setup and execution of write.
Resides in file 'sys2.c'; and calls rdwr.

writei(*ip)

Writes u.u_count bytes to file or device pointed to by inode 'ip'. Will write as many blocks as are required to exhaust the count, run out of memory, or fill the device.
Resides in file 'rdwri.c'; and calls ldiv, min, bmap, getblk, bread, iomove, brelse, bawrite, bdwrite, lshift, and dpcmp. Structures used are: inode(r,m), cdevsw(r), and user(r,m).

writep(*fp)

Internal routine to write the pipe described by 'file' structure pointed to by 'fp'. If the pipe is full then the process will wait until it is emptied by a readp. After a chunk of data is written into the pipe the reading process will be awakened if sleeping.
Resides in file 'pipe.c'; and calls plock, prele, writei, min, and wakeup. Structures used are: inode(r,m), file(r) and user(r,m).

xalloc(*ip)

Internal routine to set up text segment of a process. If it is already being used then it is a simple matter to update the use counter; otherwise, space in the text table must be found, initialized, and a copy of the text swapped out.
Resides in file 'text.c'; and calls panic, malloc, expand, estabur, readi, swap, savu, xswap, and swtch. Structures used are: text(r,m), proc(r,m), and user(r,m).

xccdec(*xp)

Decrements the count of the number of process' in main store which are using the text segment pointed to by 'xp'. If count goes to zero, the segment is removed from main store.
Resides in file 'text.c'; and calls mfree. The only structure used is text(r,m).

xfree()

Internal routine to free links to a text segment when a process terminates. If this user was the last user of a text segment then the segment is removed from the swapping disk and main memory.
Resides in file 'text.c'; and calls xccdec, mfree, and iput. Structures used are: user(r), proc(m), and text(r).

xswap(*p,flag,size)

Internal routine to swap out a process pointed to by 'p', of size 'size' (or p_size if 'size'=0). After allocating space on disk, process is written to swap device and the main store it occupied is released. Process scheduling time, p_time, is set to zero (just paged out) and process address, p_addr, is set to disk address, finally process flag, p_flag, is set to not loaded. The variable 'runout' is waked up to inform 'sched()' that there is more memory space available now.
Resides in file 'text.c'; and calls malloc, panic, xccdec, swap, mfree, and wakeup. Structures used are: proc(r,m), and runout(r,m).